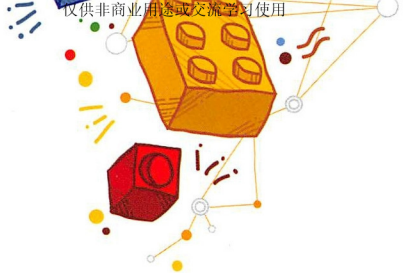


版权相关注意事项：

- 1、书籍版权归著者和出版社所有
- 2、本PDF来自于各个广泛的信息平台，经过整理而成
- 3、本PDF仅限用于非商业用途或者个人交流研究学习使用
- 4、本PDF获得者不得在互联网上以任何目的进行传播
- 5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
- 6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
- 7、请于下载PDF后24小时内研究使用并删掉本PDF



图解深度学习与神经网络

从张量到TensorFlow实现

张平 编著



中国工信出版集团



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn



图解深度学习与 神经网络

从张量到TensorFlow实现

张平 编著

电子工业出版社

Publishing House of Electronics Industry

北京•BEIJING



内 容 简 介

本书是以 TensorFlow 为工具介绍神经网络和深度学习的入门书,内容循序渐进,以简单示例和图例的形式,展示神经网络和深度学习背后的数学基础原理,帮助读者更好地理解复杂抽象的公式。同时,采用手动计算和程序代码这两种方式讲解示例,可以更好地帮助读者理解 TensorFlow 的常用函数接口,为读者掌握利用 TensorFlow 搭建人工智能项目打下良好的基础。

本书适合神经网络、深度学习、TensorFlow 的入门者阅读。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有,侵权必究。

图书在版编目(CIP)数据

图解深度学习与神经网络:从张量到 TensorFlow 实现 / 张平编著. —北京:电子工业出版社, 2018.10
ISBN 978-7-121-34745-0

I. ①图… II. ①张… III. ①学习系统—研究 ②人工神经网络—研究 ③人工智能—算法—研究
IV. ①TP273 ②TP18

中国版本图书馆 CIP 数据核字(2018)第 157982 号

策划编辑:郑柳洁

责任编辑:郑柳洁

印 刷:三河市华成印务有限公司

装 订:三河市华成印务有限公司

出版发行:电子工业出版社

北京市海淀区万寿路 173 信箱 邮编:100036

开 本:787×980 1/16 印张:21 字数:367.2 千字

版 次:2018 年 10 月第 1 版

印 次:2018 年 10 月第 1 次印刷

定 价:79.00 元

凡所购买电子工业出版社图书有缺损问题,请向购买书店调换。若书店售缺,请与本社发行部联系,联系及邮购电话:(010) 88254888, 88258888。

质量投诉请发邮件至 zltz@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式:(010) 51260888-819, faq@phei.com.cn。



前言

2016 年是人工智能进入大众视野的一年，从 AlphaGo 到无人驾驶，从量子计算机到马斯克的太空计划，每一个焦点事件的背后都与人工智能有着很大的联系。2016 年至今，短短两年的时间，人工智能在与人类生活息息相关的医疗健康、金融、零售、娱乐等方面，发挥出了巨大的潜能。

从应用领域来看，人工智能可应用于机器视觉、指纹识别、人脸识别、视网膜识别、虹膜识别、掌纹识别、专家系统、自动规划、智能搜索、语音识别、自动程序设计、智能控制、机器翻译、智能对话机器人等领域。掀起这股人工智能热潮最重要的技术之一就是深度学习（Deep Learning）技术。

本书的整体架构

本书由 14 章组成。第 1 章和第 2 章主要介绍 TensorFlow 的安装和基础使用知识。第 3 章主要介绍高等代数中的梯度和求解最优化问题的梯度下降法。第 4 章介绍如何使用第 1 章~第 3 章中讲的知识解决机器学习中比较简单的回归问题，便于读者学习后续章节中全连接神经网络和卷积神经网络的知识。第 5 章和第 6 章主要介绍全连接神经网络，包括全连接神经网络的计算步骤，如何利用全连接神经网络构建分类模型，以及全连接神经网络的梯度反向传播算法，等等。第 7 章主要介绍一维离散卷积。第 8 章主要介绍二维离散卷积。第 9 章主要介绍池化操作。第 10 章主要介绍经典的卷积神经网络。第 11 章~第 13 章主要介绍卷积神经网络的梯度反向传播。第 14 章介绍搭建神经网络的主要函数。本书中的每一章都会涉及大量的技术细节描述，便于读者掌握深度学习背后的基础知识及技术细节。

本书特色

众所周知，掌握机器学习理论的数学门槛比较高，而大量优秀的机器学习、深度学习开源框架在工程实现上帮助我们越过了这些数学细节，所以很多深度学习的相关书籍是以讲解项目为主要目标。本书试图从另一个角度引导入门者直接面对深度学习背后的数学基础，并进行了以下两点尝试：

- （1）不同书籍对同一个数学公式的符号表达可能不同，这给入门者带来了比较大的困扰。本书试图通过简单的示例和图例的形式展示复杂抽象的数学公式背后的计算原理，通过示例更好地理解复杂抽象的公式。
- （2）作者采用手动计算和利用程序代码进行处理这两种方式讲解示例，两种方式的结果可以相互验证，帮助入门者更好地理解开源框架中的函数接口。

作者认为，想在人工智能路上走得更远，内功扎实是致胜关键。希望本书可以帮助入门者夯实基础。



前言

本书面向的读者

本书的目标读者是想学习神经网络和深度学习的初学者。同时，本书的示例代码基于 TensorFlow 的 Python API，所以需要读者具备基本的 Python 编程基础。

致谢

感谢我的父母、姐姐一家人一直以来对我生活和工作的支持。

感谢 TensorFlow 开源库的所有贡献者。

感谢电子工业出版社博文视点的郑柳洁老师，在本书写作的过程中，不厌其烦地解答我遇到的各种问题，感谢她一直以来的支持和肯定。

我们期待您的反馈

限于篇幅，加之作者水平有限，书中疏漏和错误之处在所难免，恳请读者批评并指正，我们视读者的满意为己任，视读者的反馈意见为无价之宝，如果您发现了错误或者对书中内容有任何建议，都可以将其发送至电子邮箱 wxcdzhangping@126.com，也可以登录博文视点官网，在本书页面上留言。本书中所有样例的代码，均可从博文视点官网下载。

作者：张平



目录

1	深度学习及 TensorFlow 简介	1
1.1	深度学习	1
1.2	TensorFlow 简介及安装	2
2	基本的数据结构及运算	6
2.1	张量	6
2.1.1	张量的定义	6
2.1.2	Tensor 与 Numpy 的 ndarray 转换	9
2.1.3	张量的尺寸	10
2.1.4	图像转换为张量	13
2.2	随机数	14
2.2.1	均匀（平均）分布随机数	14
2.2.2	正态（高斯）分布随机数	15
2.3	单个张量的运算	17
2.3.1	改变张量的数据类型	17
2.3.2	访问张量中某一个区域的值	19
2.3.3	转置	22
2.3.4	改变形状	26
2.3.5	归约运算：求和、平均值、最大（小）值	29
2.3.6	最大（小）值的位置索引	34
2.4	多个张量之间的运算	35
2.4.1	基本运算：加、减、乘、除	35
2.4.2	乘法	41
2.4.3	张量的连接	42
2.4.4	张量的堆叠	44
2.4.5	张量的对比	48



目录

2.5	占位符	49
2.6	Variable 对象	50
3	梯度及梯度下降法	52
3.1	梯度	52
3.2	导数计算的链式法则	53
3.2.1	多个函数和的导数	54
3.2.2	复合函数的导数	54
3.2.3	单变量函数的驻点、极值点、鞍点	55
3.2.4	多变量函数的驻点、极值点、鞍点	57
3.2.5	函数的泰勒级数展开	60
3.2.6	梯度下降法	63
3.3	梯度下降法	73
3.3.1	Adagrad 法	73
3.3.2	Momentum 法	75
3.3.3	NAG 法	77
3.3.4	RMSprop 法	78
3.3.5	具备动量的 RMSprop 法	80
3.3.6	Adadelta 法	81
3.3.7	Adam 法	82
3.3.8	Batch 梯度下降	84
3.3.9	随机梯度下降	85
3.3.10	mini-Batch 梯度下降	86
3.4	参考文献	86
4	回归分析	88
4.1	线性回归分析	88
4.1.1	一元线性回归	88
4.1.2	保存和加载回归模型	91
4.1.3	多元线性回归	95
4.2	非线性回归分析	99
5	全连接神经网络	102
5.1	基本概念	102



5.2	计算步骤	104
5.3	神经网络的矩阵表达	107
5.4	激活函数	112
5.4.1	sigmoid 激活函数	112
5.4.2	tanh 激活函数	113
5.4.3	ReLU 激活函数	114
5.4.4	leaky relu 激活函数	115
5.4.5	elu 激活函数	118
5.4.6	crelu 激活函数	119
5.4.7	selu 激活函数	120
5.4.8	relu6 激活函数	121
5.4.9	softplus 激活函数	121
5.4.10	softsign 激活函数	122
5.5	参考文献	123
6	神经网络处理分类问题	125
6.1	TFRecord 文件	125
6.1.1	将 ndarray 写入 TFRecord 文件	125
6.1.2	从 TFRecord 解析数据	128
6.2	建立分类问题的数学模型	134
6.2.1	数据类别（标签）	134
6.2.2	图像与 TFRecord	135
6.2.3	建立模型	140
6.3	损失函数与训练模型	143
6.3.1	sigmoid 损失函数	143
6.3.2	softmax 损失函数	144
6.3.3	训练和评估模型	148
6.4	全连接神经网络的梯度反向传播	151
6.4.1	数学原理及示例	151
6.4.2	梯度消失	166
7	一维离散卷积	168
7.1	一维离散卷积的计算原理	168
7.1.1	full 卷积	169



7.1.2	valid 卷积	170
7.1.3	same 卷积	170
7.1.4	full、same、valid 卷积的关系	171
7.2	一维卷积定理	174
7.2.1	一维离散傅里叶变换	174
7.2.2	卷积定理	177
7.3	具备深度的一维离散卷积	182
7.3.1	具备深度的张量与卷积核的卷积	182
7.3.2	具备深度的张量分别与多个卷积核的卷积	183
7.3.3	多个具备深度的张量分别与多个卷积核的卷积	185
8	二维离散卷积	187
8.1	二维离散卷积的计算原理	187
8.1.1	full 卷积	187
8.1.2	same 卷积	189
8.1.3	valid 卷积	191
8.1.4	full、same、valid 卷积的关系	192
8.1.5	卷积结果的输出尺寸	193
8.2	离散卷积的性质	194
8.2.1	可分离的卷积核	194
8.2.2	full 和 same 卷积的性质	195
8.2.3	快速计算卷积	197
8.3	二维卷积定理	198
8.3.1	二维离散傅里叶变换	198
8.3.2	二维与一维傅里叶变换的关系	201
8.3.3	卷积定理	203
8.3.4	利用卷积定理快速计算卷积	203
8.4	多深度的离散卷积	205
8.4.1	基本的多深度卷积	205
8.4.2	1 个张量与多个卷积核的卷积	207
8.4.3	多个张量分别与多个卷积核的卷积	208
8.4.4	在每一深度上分别卷积	211
8.4.5	单个张量与多个卷积核在深度上分别卷积	212



8.4.6	分离卷积	214
9	池化操作	218
9.1	same 池化	218
9.1.1	same 最大值池化	218
9.1.2	多深度张量的 same 池化	221
9.1.3	多个三维张量的 same 最大值池化	223
9.1.4	same 平均值池化	224
9.2	valid 池化	226
9.2.1	多深度张量的 valid 池化	228
9.2.2	多个三维张量的 valid 池化	229
10	卷积神经网络	231
10.1	浅层卷积神经网络	231
10.2	LeNet	238
10.3	AlexNet	244
10.3.1	AlexNet 网络结构详解	244
10.3.2	dropout 及其梯度下降	247
10.4	VGGNet	256
10.5	GoogleNet	264
10.5.1	网中网结构	264
10.5.2	Batch Normalization	269
10.5.3	BN 与卷积运算的关系	273
10.5.4	指数移动平均	275
10.5.5	带有 BN 操作的卷积神经网络	276
10.6	ResNet	281
10.7	参考文献	284
11	卷积的梯度反向传播	286
11.1	valid 卷积的梯度	286
11.1.1	已知卷积核, 对未知张量求导	286
11.1.2	已知输入张量, 对未知卷积核求导	290
11.2	same 卷积的梯度	294
11.2.1	已知卷积核, 对输入张量求导	294



11.2.2 已知输入张量，对未知卷积核求导 298

12 池化操作的梯度 303

12.1 平均值池化的梯度 303

12.2 最大值池化的梯度 306

13 BN 的梯度反向传播 311

13.1 BN 操作与卷积的关系 311

13.2 示例详解 314

14 TensorFlow 搭建神经网络的主要函数 324



1

深度学习及 TensorFlow 简介

1.1 深度学习

2016 年是人工智能元年，随后各行各业开始全面拥抱人工智能，掀起这股热潮最重要的技术之一就是深度学习（Deep Learning）技术，它是机器学习研究中一个全新的领域。简单说，“深度学习”就是多层的神经网络，“深度”某种意义上指人工神经网络的层数，旨在模拟人的思维过程和智能行为，如学习、推理、思考、规划等，让机器的行为看起来像人所表现出的智能行为一样，甚至有可能超过人的智能。让机器人像人类一样听懂语言，就是现在的语音识别要达到的效果；让机器人像人类一样看懂文字，就是现在的自然语言处理、机器翻译、聊天机器人等要达到的效果；让机器人像人类一样看懂图片，就是现在的计算机视觉识别要达到的效果；让机器人像人类一样进行运动控制，就是现在的自动驾驶、自主行动的机器人等要达到的效果。

深度学习颠覆了语音识别、图像分类、文本理解等众多领域的算法设计思路，提供了一种直接从数据出发（输入端），经过网络结构模型得到最终结果（输出端）的端到端（end-to-end）的新模式，带动了人工智能的崛起，传统的人工智能算法是依赖人工总结的规律进行编程来解决问题的，但是深度学习有所不同，它不需要人为提取解决问题的特征或者规律，它能从输入的大量数据中自发地总结出规律，自适应地调整自身结构，从而举一反三，泛化至从未见过的案例中。用一句话来概括，深度学习最重要的特点就是能自动地从数据中学习。

神经网络背后的数学理论大概出现在 20 世纪 60 年代，至今也差不多研究了半个世纪。

回望过去，有三大壁垒阻碍了它的发展和应用：数据量和规模量小、计算机运算能力差、缺乏有效且快速的训练网络的算法。如今，我们正处于信息爆炸的大数据时代，各种数据从无穷无尽的渠道不断涌入，对于深度学习算法，获取足够多的数据，就有机会产生更好的结果。与此同时，计算机的性能也在持续提升，以前计算机的运算核心 CPU 专为顺序串行计算设计，而主流的神经网络算法类似人脑神经，不仅结构是并行的，而且处理过程也是并行且同时的，所以大规模并行处理是神经网络计算的重要特征，CPU 无法快速应对算法的复杂度带来的巨大并行运算量，但是得益于计算机游戏的图形处理单元（GPU）的快速发展，为深度学习提供了必需的计算能力，帮助拉开了深度学习迅猛发展的序幕。

常用的深度学习的三种学习模式为：

- （1）用有标注的大数据做深度监督学习。
- （2）用无标注的（更）大数据做非监督学习。
- （3）以奖励反馈为本的深度增强学习，本书主要介绍的是第一种学习模式。

1.2 TensorFlow 简介及安装

TensorFlow 是从谷歌内部使用的第一代深度学习工具 DistBelief 的经验和教训总结来的，于 2015 年 11 月作为一个新的项目向公众开源，伴随着 2016 年人工智能浪潮的到来，迅速成为计算机视觉、语音识别、机器翻译、自然语言处理等各个领域使用的深度学习工具。

TensorFlow 的安装不复杂，作者以在 Ubuntu 系统环境下，基于 Anaconda 3 的安装为例进行介绍。Anaconda 3 集成了 Python 科学计算、计算机视觉等方面的第三方库，并组织好了各个库之间的依赖关系，安装方便、快捷。首先进入官方网站，根据自己的系统和需求下载相应的版本，作者以下载 Linux 下基于 Python 3.6 的版本为例进行介绍，如图 1-1 所示。

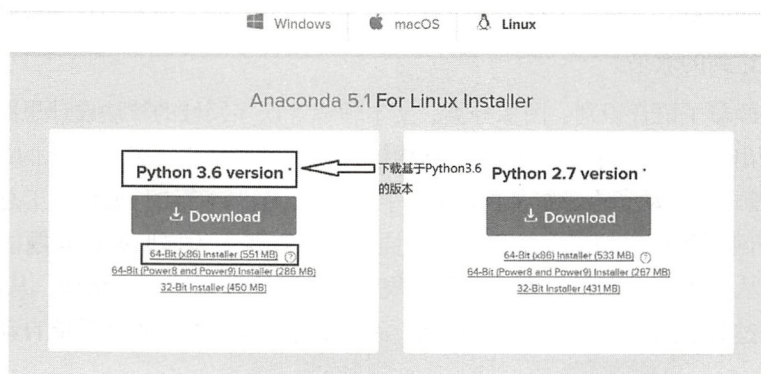


图 1-1 下载 Anaconda 3-5.1

单击“Download”按钮，下载文件 Anaconda3-5.1.0-Linux-x86_64.sh（注意：不同的版本文件名称不同），然后在终端输入以下命令，开始安装：

```
bash Anaconda3-5.1.0-Linux-x86_64.sh
```

默认安装在根目录/anaconda3 文件夹下，安装过程中会提示需要添加到系统环境变量，如图 1-2 所示，选择“yes”选项，否则，安装完成后，还需要手动添加。

```
Do you wish the installer to prepend the Anaconda3 install location
to PATH in your /home/tensorflow/.bashrc ? [yes|no]
[no] >>> 
```

图 1-2 Anaconda 3 的安装过程

安装成功后，界面如图 1-3 所示。

```
installing: jupyter-1.0.0-py36_4 ...
installing: anaconda-5.1.0-py36_2 ...
installing: conda-4.4.10-py36_0 ...
installing: conda-build-3.4.1-py36_0 ...
installation finished.
```

图 1-3 Anaconda 3 安装成功

通过网址<https://github.com/tensorflow/tensorflow>，下载对应的 TensorFlow 安装版本，如图 1-4 所示。

Individual whl files

- Linux CPU-only: Python 2 (build history) / Python 3.4 (build history) / Python 3.5 (build history) / Python 3.6 (build history)
- Linux GPU: Python 2 (build history) / Python 3.4 (build history) / Python 3.5 (build history) / Python 3.6 (build history)
- Mac CPU-only: Python 2 (build history) / Python 3 (build history)
- Windows CPU-only: Python 3.5 64-bit (build history) / Python 3.6 64-bit (build history)

图 1-4 TensorFlow 的各个安装版本

以下载 Linux CPU-only Python 3.6 (build history) 为例，下载文件 tf_nightly-1.head-cp36-cp36m-linux_x86_64.whl（注意：不同版本文件名称不同），然后在终端输入以下命令：

```
pip install tf_nightly-1.head-cp36-cp36m-linux_x86_64.whl
```

开始安装 TensorFlow，安装的过程中会自动下载各种依赖包，过程如图 1-5 所示。

图解深度学习与神经网络：从张量到 TensorFlow 实现

```
Collecting tb-nightly<1.9.0a0,>=1.8.0a0 (from tf-nightly==1.8.0)
  Downloading https://files.pythonhosted.org/packages/20/56/e5059003e85557431e04245e7c5346bb6a61b695c1fbf5aa013593430b89/tb_nightly-1.8.0a20180423-py3-none-any.whl (3.1MB)
    100% |████████████████████████████████████████| 3.1MB 127kB/s
Collecting astor>=0.6.0 (from tf-nightly==1.8.0)
  Downloading https://files.pythonhosted.org/packages/b2/91/cc9805f1ff7b49f620136b3a7ca26f6a1be2ed424606804b0fbcf499f712/astor-0.6.2-py2.py3-none-any.whl
Collecting grpcio>=1.8.6 (from tf-nightly==1.8.0)
  Downloading https://files.pythonhosted.org/packages/c8/b8/00e703183b7ae5e02f161dafacdfa8edbd7234cb7434aef00f126a3a511e/grpcio-1.11.0-cp36-cp36m-manylinux1_x86_64.whl (8.8MB)
    46% |██████████████████████████████████████| 4.1MB 126kB/s eta 0:00:38
```

图 1-5 TensorFlow 的安装过程

安装成功后，界面如图 1-6 所示。

```
Successfully installed absl-py-0.2.0 astor-0.6.2 bleach-1.5.0 gast-0.2.0 grpcio-1.11.0 html5lib-0.9999999 markdown-2.6.11 protobuf-3.5.2.post1 tb-nightly-1.8.0a20180423 termcolor-1.1.0 tf-nightly-1.9.0.dev20180417
```

图 1-6 成功安装 TensorFlow

安装完成后，在终端输入以下命令，如图 1-7 所示。

```
$ spyder
```

图 1-7 打开 Spyder 编译器

打开 Spyder 编译器，显示界面如图 1-8 所示。

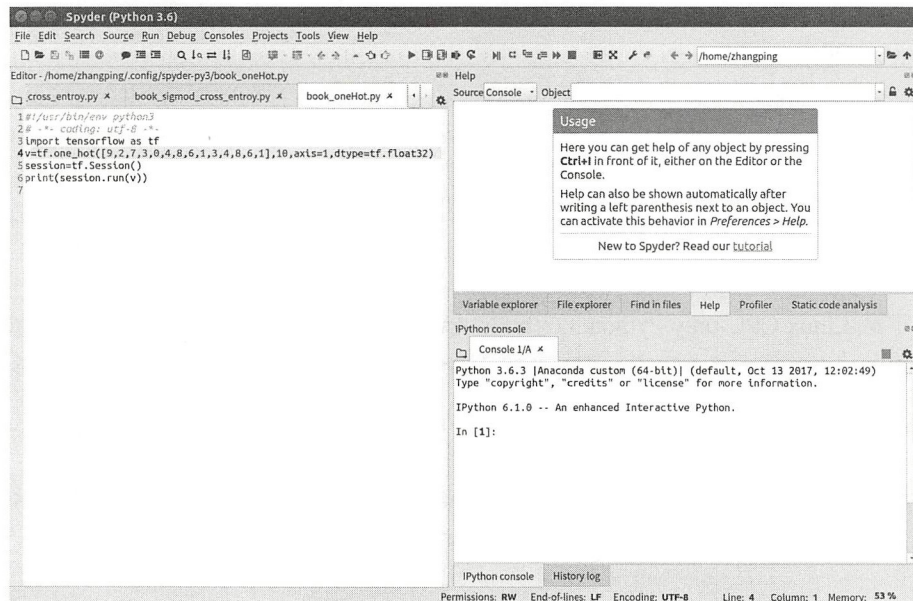


图 1-8 Spyder 界面

然后在界面右侧的命令行窗口中输入以下代码，用来打印安装的 TensorFlow 的版本号：

```
import tensorflow as tf
print(tf.__version__)
```

打印结果为：

1.5.0

代表安装的 TensorFlow 版本为 1.5，如图 1-9 所示。

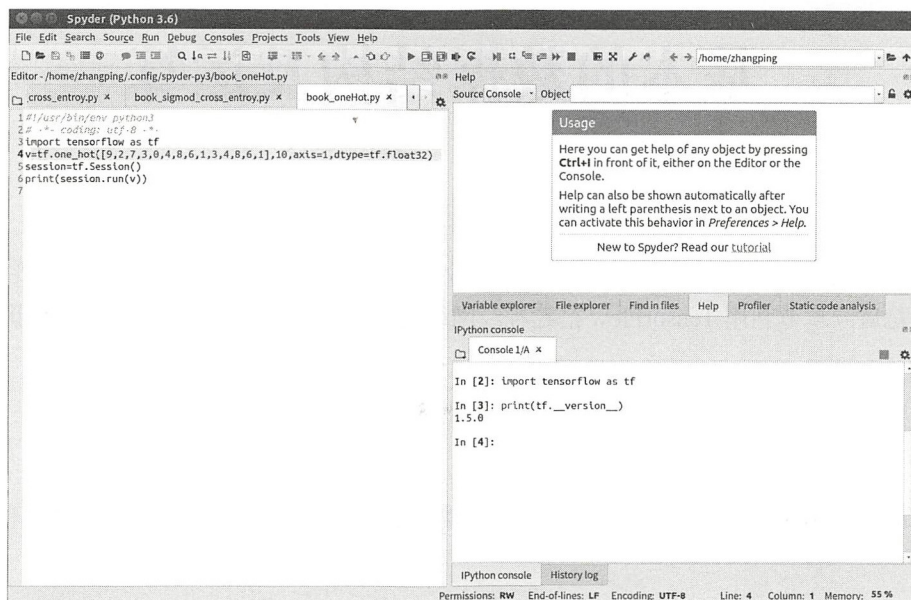


图 1-9 打印 TensorFlow 的版本号

安装 TensorFlow 其他版本的操作类似，本书不再赘述（注意：安装方法随着版本不断变化。如果读者发现按照书上的步骤安装报错，请登录 TensorFlow 官网，找到相应的版本安装步骤对照操作即可）。在成功搭建 TensorFlow 的开发环境后，第 2 章将介绍 TensorFlow 的 Python API 的核心数据结构及其基础函数，为后续章节利用 TensorFlow 学习深度学习打下基础。

2

基本的数据结构及运算

本章我们主要介绍 TensorFlow 中常用的数据结构，以及其对应的运算及函数接口，这些内容是后续章节利用 TensorFlow 搭建神经网络的基础。我们先从基本的数据结构讲起。

2.1 张量

Tensor 是 TensorFlow 中最基础、最重要的数据结构，常翻译为张量，是管理数据的一种形式。

2.1.1 张量的定义

所谓张量，可以理解为 n 维数组或者矩阵，TensorFlow 提供函数：

```
constant(value, dtype=None, shape=None, name="Const", verify_shape=False)
```

来构造张量。接下来，作者依次介绍零维、一维、二维、三维、四维张量构造的示例。

1. 零维张量

可以将零维张量理解为标量或者常数。如图 2-1 所示，用 Tensor 可将常数 3 表示为如下形式。

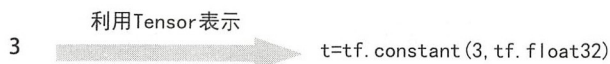


图 2-1 零维张量

2. 一维张量

可以将一维张量理解为向量。图 2-2 所示为长度是 4 的向量，可利用 Tensor 表示为如下形式。

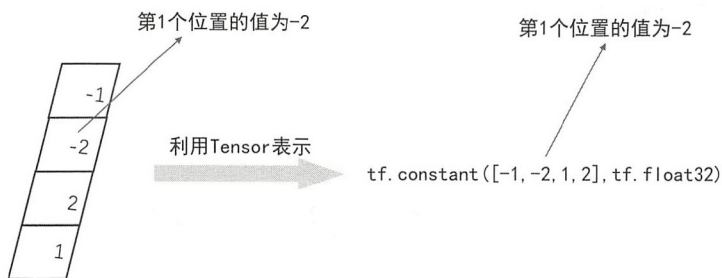


图 2-2 一维张量

3. 二维张量

可以将二维张量理解为高等数学中的矩阵。以图 2-3 所示的 3 行 4 列的矩阵为例，利用 Tensor 可以表示为如下形式。

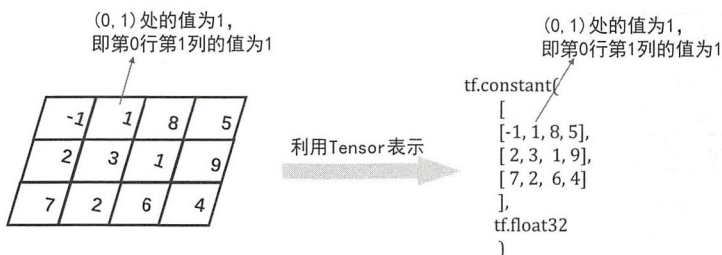


图 2-3 二维张量

4. 三维张量

可以将三维张量理解为三维数组。图 2-4 所示为 3 行 4 列 2 深度的三维张量。

三维张量可以理解为多个二维张量在深度方向的组合。对于二维张量，从图 2-3 中可知，在 (0,0) 处的值为 -1；对于三维张量，如图 2-4 所示，在 (0,0) 处的值可以理解为一个向量 [-1, -11]，这样在利用函数 `constant` 构造三维张量时会更容易理解。

图解深度学习与神经网络：从张量到 TensorFlow 实现

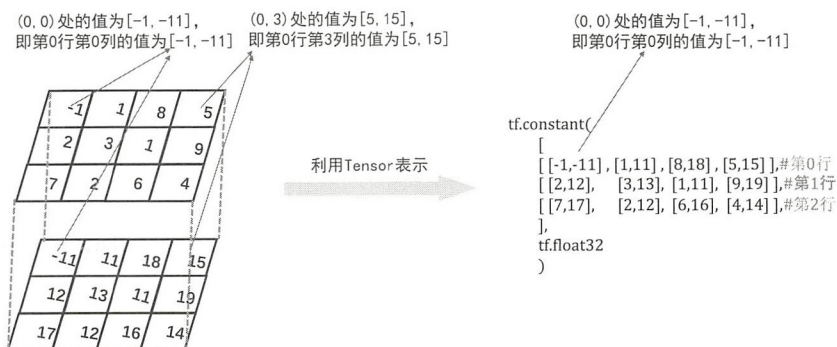


图 2-4 三维张量

5. 四维张量

可以将四维张量理解为多个三维张量。如图 2-5 所示，将 2 个三维张量视为一个整体，就是 1 个四维张量。

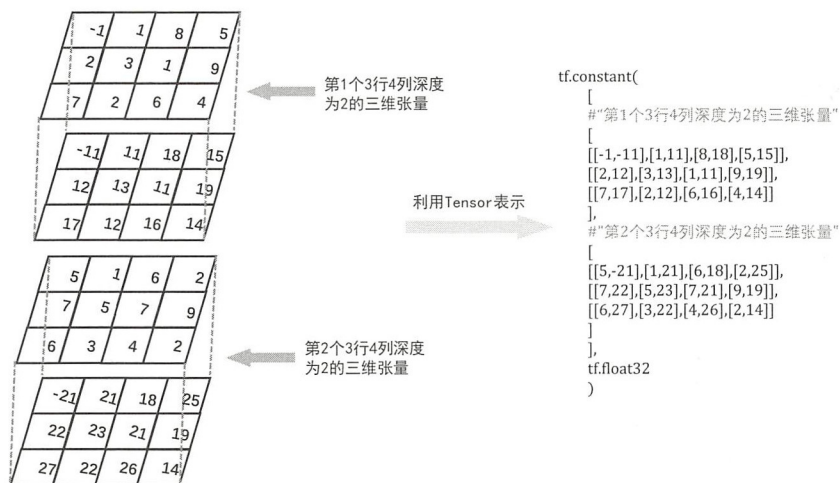


图 2-5 四维张量

理解了张量，我们将打印张量作为学习 TensorFlow 的第一步。以函数 `print` 打印一个长度为 3 的一维张量为例，它的示例代码如下：

```
import tensorflow as tf
# "一维张量"
t=tf.constant([1,2,3],tf.float32,name='t')
print(t)
```

打印结果如下：

```
Tensor("t:0", shape=(3,), dtype=float32)
```

从打印结果可以看出，并没有打印出張量的值，只是打印了张量的一些信息，如该张量的尺寸（或者称为形状）`shape=(3,)`，张量的数据类型 `dtype=float32`。关于张量的尺寸，后面的章节中会详细描述，下面介绍如何打印张量中的值。

2.1.2 Tensor 与 Numpy 的 ndarray 转换

在介绍如何打印张量的值之前，作者先简单介绍 Numpy，它是 Python 的一种开源数值计算扩展，用来存储和处理多维数组，其核心数据结构为 ndarray。TensorFlow 通过创建会话（session），将张量转换为 Numpy 中的 ndarray，才可以打印出張量的值。以下依次介绍 Tensor 和 ndarray 的相互转换。

1. Tensor 转换为 ndarray

将 Tensor 转换为 ndarray，可以通过以下具体代码理解：将一个长度为 3 的一维张量转换为 ndarray，然后打印其值。

```
# -*- coding: utf-8 -*-
import tensorflow as tf
#"一维张量"
t=tf.constant([1,2,3],tf.float32)
#"创建会话"
session=tf.Session()
#"张量转换为ndarray"
array=session.run(t)
#"打印其数据结构类型及对应的值"
print(type(array))
print(array)
```

打印结果如下：

```
<class 'numpy.ndarray'>
[1. 2. 3.]
```

也可以先创建会话，然后利用 Tensor 的成员函数 `eval`，将 Tensor 转换为 ndarray，代码如下：

图解深度学习与神经网络：从张量到 TensorFlow 实现

```
session=tf.Session()
array=t.eval(session=session)
print(array)
```

以上代码的另一种写法如下：

```
with tf.Session() as session:
    array=t.eval()
    print(array)
```

以上我们介绍了 Tensor 如何转换为 ndarray，接下来介绍 ndarray 如何转换为 Tensor。

2. ndarray 转换为 Tensor

TensorFlow 通过函数 `convert_to_tensor` 实现将 ndarray 转换为 Tensor 的功能，示例代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
import numpy as np
#"一维的ndarray"
array=np.array([1,2,3],np.float32)
#"ndarray转换为tensor"
t=tf.convert_to_tensor(array,tf.float32,name='t')
#"打印张量"
print(t)
```

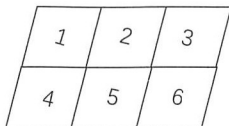
打印结果如下：

```
Tensor("t_2:0", shape=(3,), dtype=float32)
```

从打印结果可以看出，打印了张量的数据类型及其尺寸（`shape`）。接下来介绍如何获取张量的尺寸。

2.1.3 张量的尺寸

张量的尺寸，又称张量的形状，图 2-6 所示为二维张量。



1	2	3
4	5	6

图 2-6 二维张量

它的尺寸为 2 行 3 列。我们经常使用两种方式获取张量的尺寸，以下依次介绍。

1. 利用函数 `shape` 得到张量的尺寸

可以利用 TensorFlow 中的函数 `shape` 得到张量的尺寸，该函数使用方法的具体示例代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
#"张量"
t=tf.constant(
    [
        [1,2,3],
        [4,5,6]
    ],tf.float32)
session=tf.Session()
#"张量的形状"
s=tf.shape(t)
print('张量的形状:',session.run(s))
```

打印结果如下：

```
"张量的形状：" [2 3]
```

从上述代码中可以看出，利用函数 `shape` 得到张量的尺寸，需要先创建会话，然后转换为 `ndarray`，才能打印其值。下面作者介绍不需要创建会话，也可以打印出张量尺寸的方法。

2. 利用成员函数 `get_shape()` 或者成员变量 `shape` 得到张量的尺寸

可以直接利用 `Tensor` 的成员函数 `get_shape()` 或者成员变量 `shape` 得到张量的尺寸，具体示例代码如下：


```
# -*- coding: utf-8 -*-
import tensorflow as tf
#"二维张量"
t=tf.constant(
    [
        [1,2,3],
        [4,5,6]
    ],
    tf.float32
)
#"张量的尺寸"
#s=t.shape
s=t.get_shape()
#"打印s及其数据结构类型"
print("'s的值:')"
print(s)
print("'s的数据结构类型:')"
print(type(s))
print("'s[0]的值:')"
print(s[0])
#"打印s[0]及其数据结构类型"
print("'s[0]的数据结构类型:')"
print(type(s[0]))
#"将s[0]转换为整数型"
print("'将s[0]的值转换为整数型:')"
print(s[0].value)
print(type(s[0].value))
```

打印结果如下：

```
"s的值:"
(2, 3)
"s的数据结构类型:"
<class 'tensorflow.python.framework.tensor_shape.TensorShape'>
"s[0]的值:"
2
"s[0]的数据结构类型:"
```

```
<class 'tensorflow.python.framework.tensor_shape.Dimension'>
"将s[0]的值转换为整数型:"
2
<class 'int'>
```

我们来分析以上代码及其打印结果。首先，利用函数 `get_shape()` 或者成员变量 `shape` 返回得到值 `s`。它是一个 `TensorShape` 类，而 `s[0]` 是一个 `Dimension` 类。然后，利用其成员变量 `value` 返回其整数类型。该方式不用创建会话，使用起来比较方便。需要注意的是，可以使用负数索引，以上示例中，`s[-1]` 代表最后一个维度的维数，因为 `s` 是一个二维张量，所以最后一维代表列，倒数第二维代表行，则 `s[-1]=s[1]`、`s[-2]=s[0]`，那么什么情况下会用负数索引呢？例如，需要知道一个张量最后一个维度的维数，但又不知道这个张量是几维的，所以不能用正数索引，这时用负数索引会更方便，这些优点在后续章节的代码中会慢慢展现出来。

在 2.1.2 节中，作者已经介绍了 `ndarray` 与 `Tensor` 的相互转换，接下来介绍如何将图像数字化为张量。

2.1.4 图像转换为张量

`TensorFlow` 中提供了一些读取不同文件类型的接口，我们首先介绍读取图像文件的方法，在后续章节中会介绍读取其他文件的接口的方法，作者以读取图像文件 `test.jpg` 为例进行讲解，如图 2-7 所示。

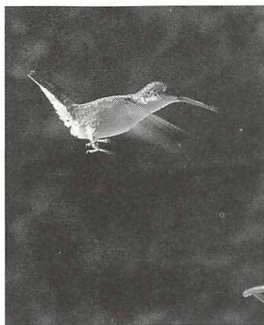


图 2-7 test.jpg

首先，通过函数 `read_file` 读取本地的图像文件。然后，利用函数 `decode_jpeg` 解码为张量。以下代码实现了将图像转换为张量并打印图像的形状，然后将张量转换为 `ndarray` 并显示图像，具体代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
import matplotlib.pyplot as plt
#"读取图像文件"
image=tf.read_file("test.jpg",'r')
#"将图像文件解码为Tensor"
image_tensor=tf.image.decode_jpeg(image)
#"图像张量的形状"
shape=tf.shape(image_tensor)
session=tf.Session()
print('图像的形状:')
print(session.run(shape))
#"Tensor转换为ndarray"
image_ndarray=image_tensor.eval(session=session)
#"显示图片"
plt.imshow(image_ndarray)
plt.show()
```

打印结果如下：

```
"图像的形状:"
[292 400   3]
```

从打印结果可以看出，TensorFlow 也将图像数字化为一个三维张量。

2.2 随机数

随机数在机器学习中起着重要的作用，TensorFlow 提供了很多产生不同概率分布的随机数的函数，均以 `random` 开头，如产生均匀分布随机数的函数 `random_uniform`、产生正态分布随机数的函数 `random_norm`、产生泊松分布随机数的函数 `random_poisson_v2` 等。本节将介绍两种最常用的随机数：均匀分布随机数和正态分布随机数。

2.2.1 均匀（平均）分布随机数

均匀分布的概率密度函数为

$$p(X = x) = \begin{cases} \frac{1}{b-a}, & a \leq x < b \\ 0, & \text{其他} \end{cases}$$

即等概率地取在区间 $[a, b]$ 中的值。我们利用函数 `random_uniform` 构造 1 个四维张量，该张量是 10 个 4 行 20 列深度为 5 的三维张量，其中的每一个值都在区间 $[0, 10]$ 内且满足均匀分布的随机数，代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
import matplotlib.pyplot as plt
#"四维张量:张量中的值满足最小值0、最大值10的均匀分布的随机数"
x=tf.random_uniform([10,4,20,5],minval=0,maxval=10,dtype=tf.float32)
session=tf.Session()
#"Tensor转换为ndarray"
array=session.run(x)
#"为了画出直方图，将array转换为1个一维的ndarray"
array1d=array.reshape([-1])
plt.hist(array1d)
plt.show()
```

运行结果如图 2-8 所示。

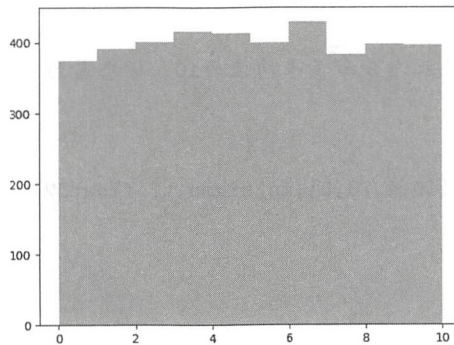


图 2-8 均匀分布的随机数的直方图

因为是随机数，所以每次运行的结果是有差别的。

2.2.2 正态（高斯）分布随机数

正态分布又称高斯分布，一元正态分布的概率密度函数为

$$P(X = x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

其中 μ 代表均值, σ 代表标准差, 如图 2-9 所示, 显示了不同均值和标准差的正态概率密度函数。

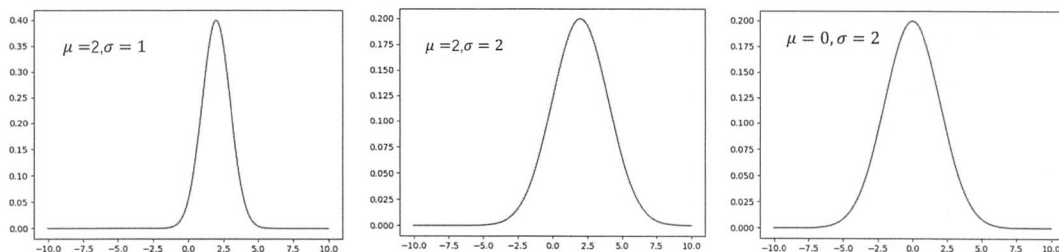


图 2-9 正态分布

接下来, 我们利用函数 `random_norm` 构造 1 个四维张量, 张量中的值都满足平均值为 10, 标准差为 1 的正态分布的随机数。然后, 将构造的张量转换为 `ndarray`, 再画出直方图, 代码如下:

```
# -*- coding: utf-8 -*-
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
import math

#"四维张量:张量中的值满足正态分布(平均值为10, 标准差为1)的随机数"
sigma=1
mu=10
result=tf.random_normal([10,4,20,5],mu,sigma,tf.float32)
session=tf.Session()
#"Tensor转换为ndarray"
array=session.run(result)
#"将多维的ndarray转换为一维的ndarray"
array1d=array.reshape([-1])
#"计算并显示直方图"
histogram,bins,patch= plt.hist(array1d,25,facecolor='gray',
                                alpha=0.5,normed=True)
x=np.arange(5,15,0.01)
y=1.0/(math.sqrt(2*np.pi)*sigma)*
    np.exp(-np.power(x-mu,2.0)/(2*math.pow(sigma,2)))
```

```
plt.plot(x,y)
plt.show()
```

运行结果如图 2-10 所示。

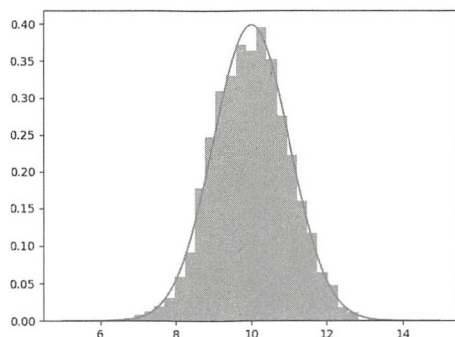


图 2-10 正态分布随机数的直方图

上述程序每次运行的结果可能不同。

至此，我们介绍了张量的构造及属性，接下来介绍张量的运算。TensorFlow 提供了非常多的运算接口，由于篇幅有限，不能一一介绍，只介绍常用的运算接口。我们把这些运算分为两大类：单个张量的运算和多个张量的运算。首先，介绍单个张量的运算。

2.3 单个张量的运算

2.3.1 改变张量的数据类型

TensorFlow 通过函数 `cast` 实现张量数据类型的转换，例如，整型与浮点型之间的转换、数值型与 `bool` 型之间的转换，等等。作者以数值型与 `bool` 型之间的转换为例，介绍该函数的使用方法。

1. 数值型转换为 `bool` 型

一个张量由数值型转换为 `bool` 型，可以用图 2-11 所示的示例理解。将一个 2 行 3 列的浮点型二维张量转换为 `bool` 型，张量中等于 0 的值转换为 `False`，不等于 0 的值转换为 `True`。

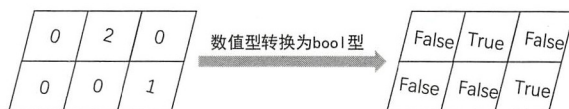


图 2-11 数值型转换为 bool 型

以上示例对应的代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
#"张量"
t=tf.constant(
    [
        [0,2,0],
        [0,0,1]
    ],tf.float32)
session=tf.Session()
#"数值型转换为bool型"
r=tf.cast(t,tf.bool)
print(session.run(r))
```

打印结果如下：

```
[[False True False]
 [False False True]]
```

2. bool 型转换为数值型

我们可以通过图 2-12 所示的示例理解如何将一个 bool 型的张量转换为数值型，其中 False 转换为 0，True 转换为 1。

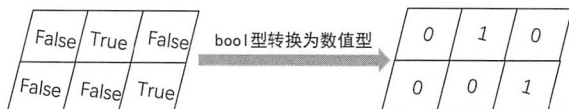


图 2-12 bool 型转换为数值型

以上示例对应的具体代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
#"张量"
t=tf.constant(
    [
        [False,True,False],
        [False,False,True]
    ],tf.bool)
session=tf.Session()
#"bool型转换为数值型"
r=tf.cast(t,tf.float32)
print(session.run(r))
```

打印结果如下：

```
[[ 0.  1.  0.]
 [ 0.  0.  1.]]
```

其他类型之间的相互转换是类似的，本节不再赘述。接下来，我们介绍如何访问张量中某一个区域的值。

2.3.2 访问张量中某一个区域的值

TensorFlow 通过函数 `slice` 访问张量中任意一个区域的值。下面我们依次介绍如何使用该函数访问一维、二维、三维张量中某一区域的值。

1. 一维张量中某一个区域的值

我们通过图 2-13 所示的示例解释如何获取一维张量中某一个区域的值，例如，获取从第 1 个位置（位置索引是从 0 开始的）开始，长度为 3 的区域的值。

以上示例对应的具体代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
#"长度为5的一维张量"
t1=tf.constant([1,2,3,4,5],tf.float32)
```



```
#"从t1的第1个位置开始，取长度为3的区域的值"  
t=tf.slice(t1,[1],[3])  
#"创建会话"  
session=tf.Session()  
#"打印结果"  
print(session.run(t))
```

打印结果如下：

```
[ 2.  3.  4.]
```

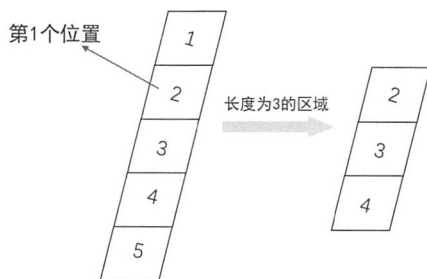


图 2-13 一维张量中从第 1 个位置开始，访问长度为 3 的区域

2. 二维张量中某一个区域的值

我们通过图 2-14 所示的示例理解如何获取二维张量中某一区域的值，例如，获取从位置 (0,1) 开始，高为 2、宽为 2 的区域的值。

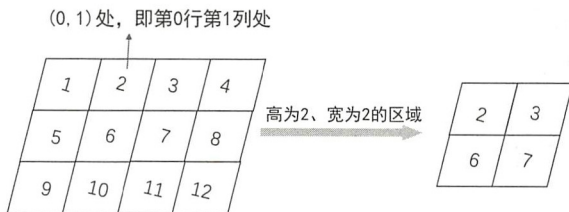


图 2-14 二维张量中的某一个区域的值

以上示例的具体代码实现如下：

```
# -*- coding: utf-8 -*-  
import tensorflow as tf  
#"3行4列的二维张量"  
t2=tf.constant(  

```

```

[
    [1,2,3,4],
    [5,6,7,8],
    [9,10,11,12]
],tf.float32
)
#"从[0,1]位置开始，取高为2、宽为2的区域的值"
t=tf.slice(t2,[0,1],[2,2])
#"创建会话"
session=tf.Session()
#"打印结果"
print(session.run(t))

```

打印结果如下：

```

[[ 2.  3.]
 [ 6.  7.]]

```

3. 三维张量中某一个区域的值

我们通过图 2-15 所示的示例介绍如何获取三维张量中某一个区域的值，例如，获取从位置 (1,0,1) 开始，即从第 1 行第 0 列第 1 深度的位置开始，高为 2、宽为 2、深度为 1 的区域的值。

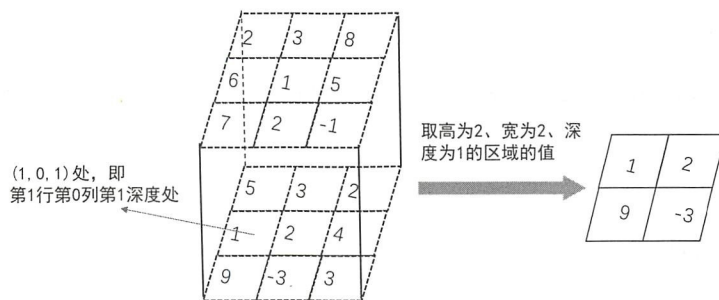


图 2-15 三维张量中的某一个区域的值

以上示例的具体代码实现如下：

```

# -*- coding: utf-8 -*-
import tensorflow as tf

```

图解深度学习与神经网络：从张量到 TensorFlow 实现

```

#"3行3列2深度的三维张量"
t3d=tf.constant(
    [
        [[2,5],[3,3],[8,2]],
        [[6,1],[1,2],[5,4]],
        [[7,9],[2,-3],[-1,3]]
    ],tf.float32
)

#"从[1,0,1]位置开始，取高为2、宽为2、深度为1的区域的值"
t=tf.slice(t3d,[1,0,1],[2,2,1])

#"创建会话"
session=tf.Session()

#"打印结果"
print(session.run(t))

```

打印结果如下：

```

[[[ 1.],[ 2.]],
 [[ 9.],[-3.]]]

```

访问更高维张量的区域的方法类似，本节不再赘述。接下来，我们介绍张量的转置操作。

2.3.3 转置

TensorFlow 提供函数 `transpose` 实现张量的转置功能，下面作者将依次介绍二维和三维张量的转置。

1. 二维张量的转置

假设有 H 行 W 列的二维张量 \mathbf{T} ，转置后为 W 行 H 列的二维张量 \mathbf{T}^T ，且

$$\mathbf{T}^T(w, h) = \mathbf{T}(h, w)$$

即 \mathbf{T} 的第 h 行第 w 列的值等于 \mathbf{T}^T 的第 w 行 h 列的值。

举例：图 2-16 所示的 2 行 3 列的二维张量转置后为 3 行 2 列的二维张量。

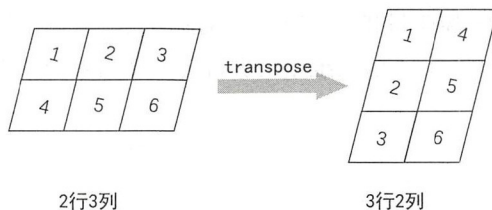


图 2-16 二维张量的转置

上述示例的对应代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
#"2行3列的二维张量"
x=tf.constant(
    [
        [1,2,3],
        [4,5,6]
    ],tf.float32
)
#"创建会话"
session = tf.Session()
#"转置"
r=tf.transpose(x,perm=[1,0])
print(session.run(r))
```

打印结果如下：

```
[[1. 2. 3.]
 [4. 5. 6.]]
```

注意：当函数 `transpose` 中的参数 `perm=[0,1]` 时，表示对张量不做任何转置操作。我们会通过三维张量的转置，详细解释以下参数 `perm` 的作用。

2. 三维张量的转置

三维张量的转置略复杂，我们以图 2-17 所示的 2 行 3 列深度为 2 的张量为例讲解。

三维张量有一个默认的坐标轴，“0”代表沿行的方向，“1”代表沿列的方向，“2”代表沿深度的方向。例如，我们常称的“深度”，也可以称为“(0,1)平面”，我们将在介绍归约运算时详细描述它的具体作用。利用函数 `transpose` 对上述三维张量进行转置操作，当参



图解深度学习与神经网络：从张量到 TensorFlow 实现

数 $\text{perm}=[0,1,2]$ 时，结果与原张量相等；当参数 $\text{perm}=[1,0,2]$ 时，代表对三维张量的每一个深度上（即每一个“(0,1)平面”）的二维张量进行转置，结果如图 2-18 所示。

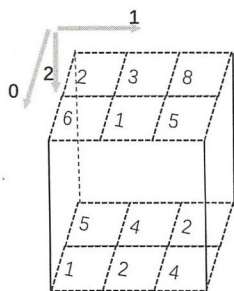
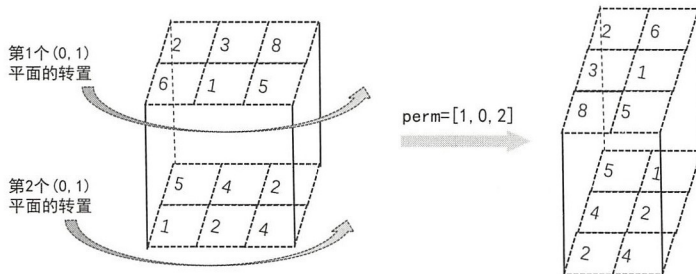


图 2-17 三维张量

图 2-18 $\text{perm}=[1,0,2]$ 时的三维张量的转置

以上示例对应的具体代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
#"2行3列2深度的三维张量"
x=tf.constant(
    [
        [[2,5],[3,4],[8,2]],
        [[6,1],[1,2],[5,4]]
    ],tf.float32
)
#"创建会话"
session = tf.Session()
#"每一个(0,1)平面的转置"
r=tf.transpose(x,perm=[1,0,2])
print(session.run(r))
```

打印结果如下：

```
[[[2. 5.]
  [6. 1.]]

 [[3. 4.]
  [1. 2.]]

 [[8. 2.]
```



[5. 4.]]]

同理，当参数 `perm=[0,2,1]` 时，代表对三维张量的每一个“(1,2)平面”上的二维张量分别进行转置，结果如图 2-19 所示。

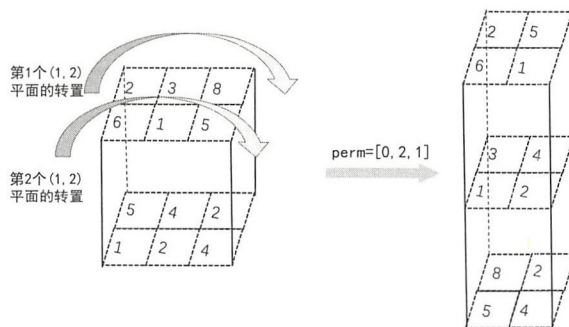


图 2-19 `perm=[0,2,1]` 时的三维张量的转置

代码如下：

```
r=tf.transpose(x,perm=[1,0,2])
```

打印结果如下：

```
[[[2. 3. 8.]
  [5. 4. 2.]]]
```

```
[[[6. 1. 5.]
  [1. 2. 4.]]]
```

同理，当参数 `perm=[2,1,0]` 时，代表对三维张量的每一个“(0,2)平面”上的二维张量分别进行转置，结果如图 2-20 所示。

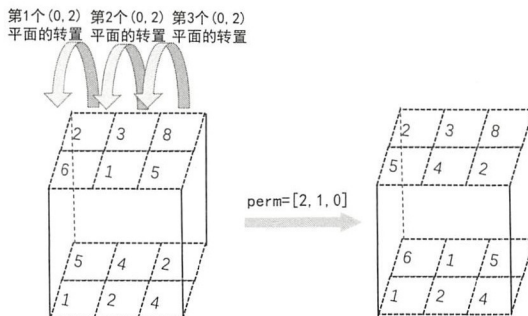


图 2-20 `perm=[2,1,0]` 时的三维张量的转置



图解深度学习与神经网络：从张量到 TensorFlow 实现

代码如下：

```
r=tf.transpose(x,perm=[2,1,0])
```

打印结果如下：

```
[[[2. 6.]
  [3. 1.]
  [8. 5.]]

 [[5. 1.]
  [4. 2.]
  [2. 4.]]]
```

以上三种转置操作是三维张量基本的转置操作，想象这些数字在一个三阶魔方上可能更容易理解，其他更复杂的情况是由这三种基本操作演变而成的。例如，当 `perm=[1,2,0]` 时，转置结果就是先让原张量在 `perm=[2,1,0]` 时转置，得到的结果接着在 `perm=[1,0,2]` 时转置即可。

在 2.1.3 节中，我们介绍了张量的形状，接下来，我们介绍如何改变张量的形状。

2.3.4 改变形状

TensorFlow 通过函数 `reshape` 改变张量的形状，通常有以下两种情况：改变张量的维数和不改变张量的维数。

1. 在不改变张量维数的基础上，改变张量的形状

我们通过图 2-21 所示的示例介绍如何在不改变张量维数的基础上，改变张量的形状，将一个三维张量转换为另一个三维张量，只是形状上从 2 行 3 列 2 深度转换为 4 行 1 列 3 深度。

以上示例的具体代码实现如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
#"2行3列2深度的三维张量"
t3d=tf.constant(
    [
        [[1,2],[4,5],[6,7]],
```



```

[[8,9],[10,11],[12,13]]
],tf.float32
)

session=tf.Session()
#"形状改变为4行1列3深度的三维张量"
t1 = tf.reshape(t3d,[4,1,-1])
#"打印结果"
print(session.run(t1))

```

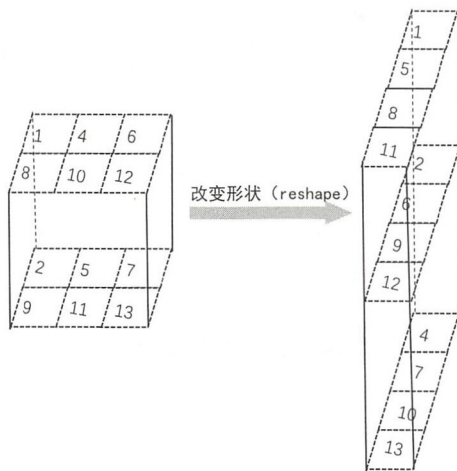


图 2-21 2 行 3 列 2 深度的三维张量转换为 4 行 1 列 3 深度的三维张量

打印结果如下：

```

[[[ 1.  2.  4.]],
 [[ 5.  6.  7.]],
 [[ 8.  9. 10.]],
 [[11. 12. 13.]]]

```

注意：以上示例程序中的 `t1 = tf.reshape(t3d,[4,1,-1])` 等价于 `t1 = tf.reshape(t3d,[4,1,3])`。

`[4, 1, -1]` 中的 `-1` 表示不用计算出这一维度上的具体维数，TensorFlow 会根据张量中总的数值个数和给出的其他维度的维数自动计算，比如在该示例中总的数值个数是 12，而改变形状后的行数是 4，列数是 1，那么深度自然是 3。虽然我们在程序中设置的是 `-1`，但是函数 `reshape` 内部会根据这些条件计算出深度方向上真实的维数。



2. 在改变张量维数的基础上，改变张量的形状

我们通过图 2-22 所示的示例讲解如何在改变张量维数的基础上，改变张量的形状。将 1 个四维张量转换为 1 个二维张量，实际上是 2 个 2 行 3 列 2 深度的三维张量转换为 1 个 2 行的二维张量（即每 1 个三维张量转换为一维张量）：

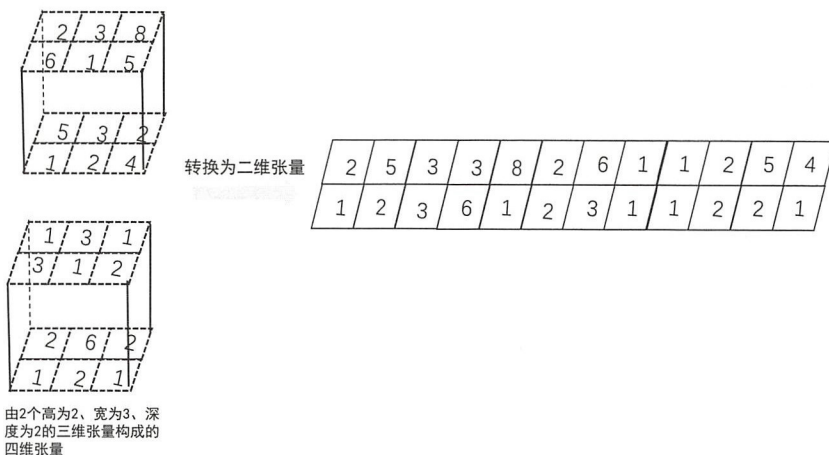


图 2-22 2 个 2 行 3 列 2 深度的三维张量转换为 1 个 2 行的二维张量

以上示例对应的具体代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
#"四维张量"
t4d=tf.constant(
    [
        #"第1个高为2、宽为3、深度为2的三维张量"
        [
            [[2,5],[3,3],[8,2]],
            [[6,1],[1,2],[5,4]]
        ],
        #"第2个高为2、宽为3、深度为2的三维张量"
        [
            [[1,2],[3,6],[1,2]],
            [[3,1],[1,2],[2,1]]
        ]
    ],tf.float32
```



```

    )
    #"转换为高为2的二维张量"
    t2d=tf.reshape(t4d,[2,-1])
    #t2d=tf.reshape(t4d,[-1,3*3*2])
    #"创建会话"
    session=tf.Session()
    #"打印结果"
    print(session.run(t2d))

```

打印结果如下：

```

[[ 2.  5.  3.  3.  8.  2.  6.  1.  1.  2.  5.  4.]
 [ 1.  2.  3.  6.  1.  2.  3.  1.  1.  2.  2.  1.]]

```

接下来，我们介绍归约运算。

2.3.5 归约运算：求和、平均值、最大（小）值

归约运算（reduction）经常被用来表示在大规模数据集中查寻数据集的总和、平均值、最大值、最小值等问题。TensorFlow 中以 `reduce` 开头的函数接口代表归约运算，如 `reduce_sum` 用于计算张量的和、`reduce_mean` 用于计算张量的平均值、`reduce_max` 用于计算张量的最大值、`reduce_min` 用于计算张量的最小值，其他函数类似。这些函数的使用方法非常类似，作者以求和运算 `reduce_sum` 为例，依次介绍一维、二维、三维张量的求和。

1. 一维张量的和

归约运算的函数大多具备一个参数 `axis`，该参数用于指明沿张量哪个维度（方向）进行归约计算。图 2-23 所示为一维张量的和，一维张量只有一个方向，即“0”方向。

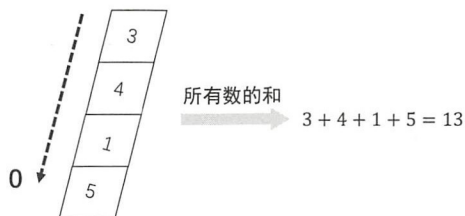


图 2-23 一维张量的和



以上示例对应的具体代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
# "一维张量"
t1d=tf.constant([3,4,1,5],tf.float32)
# "求和"
sum0=tf.reduce_sum(t1d)
# "打印结果"
session=tf.Session()
print(session.run(sum0))
```

打印结果如下：

13.0

以上代码中，`sum0=tf.reduce_sum(t1d)` 等价于 `sum0=tf.reduce_sum(t1d,axis=0)`。

2. 二维张量的和

我们以图 2-24 所示的二维张量为例，介绍二维张量的和。二维张量也有其默认的方向，其中“0”代表沿行方向，“1”代表沿列方向，“(0,1)”代表整个二维张量。

我们利用函数 `reduce_sum` 计算以上二维张量每一列的和，每一列即沿行方向，令参数 `axis=0`，其结果如图 2-25 所示。

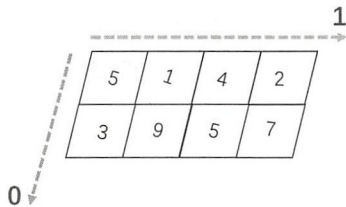


图 2-24 二维张量

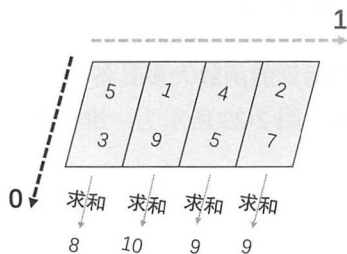


图 2-25 计算每一列的和

同理，计算以上二维张量每一行的和，每一行即沿列方向，令参数 `axis=1`，其结果如图 2-26 所示。

如果计算以上二维张量所有数的和，则令参数 `axis=(0,1)`，其结果如图 2-27 所示。



2 基本的数据结构及运算

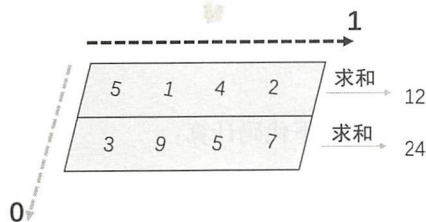


图 2-26 计算每一行的和

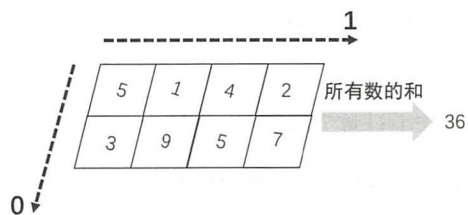


图 2-27 计算所有数的和

以上示例对应的具体代码如下：

```
import tensorflow as tf
#"二维张量"
value2d=tf.constant(
    [
        [5,1,4,2],
        [3,9,5,7]
    ],tf.float32
)
#"创建会话"
session=tf.Session()
#"计算沿"0"轴方向的和"
sum0=tf.reduce_sum(value2d,axis=0)
print("沿"0"轴方向的和:")
print(session.run(sum0))
#"计算沿"1"轴方向的和"
sum1=tf.reduce_sum(value2d,axis=1)
print("沿"1"轴方向的和:")
print(session.run(sum1))
#"计算(0,1)平面上的和"
sum01=tf.reduce_sum(value2d,axis=(0,1))
print("沿(0,1)平面上的和:")
print(session.run(sum01))
```

打印结果如下：

```
"沿"0"轴方向的和:"
[ 8. 10.  9.  9.]
"沿"1"轴方向的和:"
[ 12. 24.]
```



"沿 (0,1) 平面上的和:"

36.0

其中计算 (0,1) 平面上的和, 即所有值的和, 也可以用以下代码计算:

```
sum01=tf.reduce_sum(value2d)
```

即函数 `reduce_sum` 默认就是计算张量内所有值的和。

3. 三维张量的和

同二维张量类似, 三维张量也有默认的方向, 其中“0”代表沿行的方向, “1”代表沿列的方向, “2”代表沿深度的方向, 图 2-28 所示为 3 行 3 列 2 深度的三维张量。

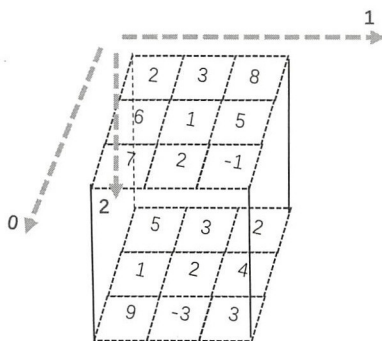


图 2-28 三维张量

我们利用函数 `reduce_sum` 计算图 2-28 所示的三维张量的和。如果令参数 `axis=0`, 则表示沿“0”方向进行归约求和, 注意这时每个深度是独立的, 计算的是每一个深度上每一列的和。如果令参数沿“1”方向进行归约运算, 结果如图 2-29 所示。

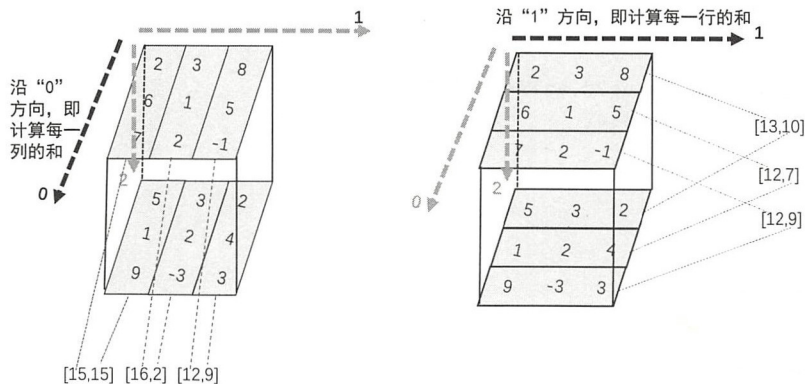


图 2-29 三维张量沿“0”和“1”方向的和



令参数 `axis=(0,1)`，代表分别计算每一深度上的和。令 `axis=(0,1,2)`，表示计算整个三维张量的和，其结果如图 2-30 所示。

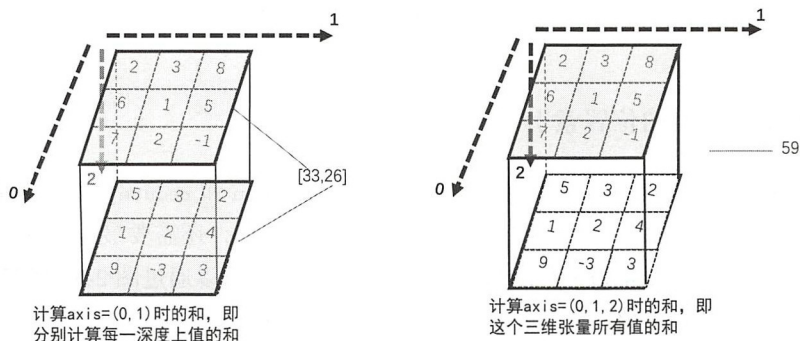


图 2-30 当 `axis=(0,1)` 或 `axis=(0,1,2)` 时，三维张量的和

以下代码只计算三维张量沿“0”方向的和，其他任意方向上，只需修改参数 `axis` 即可，具体代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
#"三维张量"
value3d=tf.constant(
    [
        [[2,5],[3,3],[8,2]],
        [[6,1],[1,2],[5,4]],
        [[7,9],[2,-3],[-1,3]]
    ],tf.float32
)
#"创建会话"
session=tf.Session()
#"计算沿"0"方向的和"
sum0=tf.reduce_sum(value3d,axis=0)
print(session.run(sum0))
```

打印结果如下：

```
[[15. 15.]
 [ 6.  2.]
 [12.  9.]]
```



从打印结果可以看出，它同图 2-29 所示的结果相等。

其他类型的归约运算如最大值、最小值、平均值等，在计算时只要改成相应的函数即可。接着，我们介绍同最大（小）值归约运算类似的一种运算：最大（小）值的位置索引。

2.3.6 最大（小）值的位置索引

在 2.3.5 节中，我们介绍了如何利用归约函数 `reduce_min` 和 `reduce_max` 计算张量中的最小值和最大值，但有时我们需要的不只是一个张量中的最值，还需要最值在张量中的位置，比如我们需要知道二维张量每一行的最大值的位置，这一点可以通过图 2-31 理解。同理，如果要知道每一列的最大值的位置，可以通过图 2-32 理解。

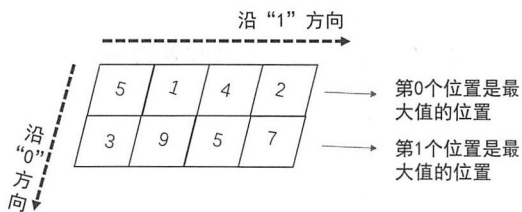


图 2-31 二维张量第一行的最大值的位置

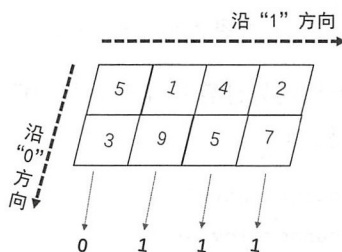


图 2-32 二维张量第一列的最大值的位置

TensorFlow 通过函数 `argmax` 计算张量中最大值的位置索引，以上示例对应的代码如下：

```
import tensorflow as tf
#"二维张量"
value2d=tf.constant(
    [
        [5,1,4,2],
        [3,9,5,7]
    ],tf.float32
)
#"求沿"1"方向上，即每一行的最大值的位置索引"
result=tf.argmax(value2d,axis=1)
#result=tf.argmax(value2d,1) 该函数已经被舍弃
session=tf.Session()
#"打印结果"
print(session.run(result))
```

打印结果如下：



[0 1]

同理，计算沿“0”方向上，即每一列的最大值的位置索引，如图 2-32 所示。

对应代码如下：

```
result=tf.argmax(value2d,axis=0)
```

打印结果如下：

[0 1 1 1]

TensorFlow 通过函数 `argmin` 实现计算张量中最小值的位置索引的功能。

以上主要介绍了单个张量的运算，接下来介绍常用的多个张量的运算。

2.4 多个张量之间的运算

提起多个张量之间的运算，最简单的莫过于加、减、乘、除这四种基本运算，以下我们就从这四个运算开始。

2.4.1 基本运算：加、减、乘、除

我们以加法为例进行讲解，其他运算类似。

1. 二维张量的加法

TensorFlow 中的函数 `add` 用来实现张量与张量之间的加法。以二维张量的加法为例，通常 M 行 N 列的二维张量只能与另一个 M 行 N 列的二维张量或者一个常数相加，但函数 `add` 可以更方便地实现其他的加法需求，如可以方便地实现一个二维张量的每一列分别和同一个列张量相加，图 2-33 所示的 2 行 3 列的二维张量可以与一个 2 行 1 列的张量相加，即每一列分别与这个 2 行 1 列的张量相加。

$$\begin{array}{|c|c|c|} \hline 1 & 2 & 3 \\ \hline 4 & 5 & 6 \\ \hline \end{array} + \begin{array}{|c|} \hline 10 \\ \hline 20 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline 11 & 12 & 13 \\ \hline 24 & 25 & 26 \\ \hline \end{array}$$

图 2-33 二维张量的和

图解深度学习与神经网络：从张量到 TensorFlow 实现

以上示例的对应代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
#"2行3列的二维张量"
value1=tf.constant(
    [
        [1,2,3],
        [4,5,6]
    ],tf.float32)
#"2行1列的二维张量"
value2=tf.constant([
    [10],
    [20]
],tf.float32)
#"加法运算"
result=tf.add(value1,value2)
#"创建会话"
session=tf.Session()
#"打印结果"
print(session.run(result))
```

打印结果如下：

```
[[ 11.  12.  13.]
 [ 24.  25.  26.]]
```

同理，令以上 2 行 3 列的二维张量的每一行与一个行张量相加，如图 2-34 所示。

$$\begin{array}{|c|c|c|} \hline 1 & 2 & 3 \\ \hline 4 & 5 & 6 \\ \hline \end{array} + \begin{array}{|c|c|c|} \hline 100 & 200 & 300 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline 101 & 202 & 303 \\ \hline 104 & 205 & 306 \\ \hline \end{array}$$

图 2-34 二维张量的每一行加上同一个行张量

只需将上述代码的 value2 替换为：

```
#"1行3列的二维张量"
value2=tf.constant([[100,200,300]],tf.float32)
```

或简化为：

```
#"长度为3的一维张量"
value2=tf.constant([100,200,300],tf.float32)
```

打印结果如下：

```
[[ 101.  202.  303.]
 [ 104.  205.  306.]]
```

理解了以上示例后，我们对函数 `add` 实现的针对二维张量的加法总结如下。

- (1) M 行 N 列的二维张量可以与另一个 M 行 N 列的二维张量相加。
- (2) M 行 N 列的二维张量可以与 M 行 1 列的二维张量相加。
- (3) M 行 N 列的二维张量可以与 1 行 N 列的二维张量相加。这种情况下，在程序实现中可以将 1 行 N 列的二维张量直接简化为一个长度是 N 的一维张量。

2. 三维张量的加法

我们以图 2-35 所示的 2 行 2 列 2 深度的三维张量 \mathbf{x} 与不同的张量 \mathbf{y} 的加法为例介绍三维张量的加法。

显然， \mathbf{x} 可以和 2 行 2 列 2 深度的三维张量 \mathbf{y} 进行加法运算，只要对应位置相加即可，如图 2-36 所示。

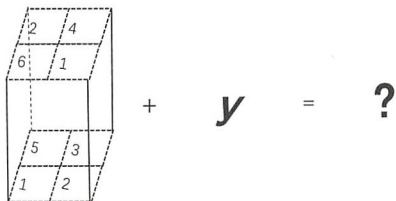


图 2-35 三维张量的加法

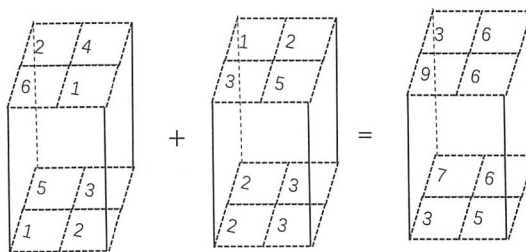


图 2-36 相同尺寸的三维张量的加法

也可以用加法运算符 “+” 替代函数 `add`，与图 2-36 对应的代码如下：

```
import tensorflow as tf
#"2行2列2深度的二维张量"
x=tf.constant(
    [
        [[2,5],[4,3]],
        [[6,1],[1,2]]
```

```

    ],tf.float32
    )
#"2行2列2深度的二维张量"
y=tf.constant(
    [
        [[1,2],[2,3]],
        [[3,2],[5,3]]
    ],tf.float32
    )

session=tf.Session()
#"x与y的和"
result=x+y
print(session.run(result))

```

x 还可以与 2 行 1 列 2 深度的三维张量 y 相加，这时我们将三维张量的每一个深度看作一个二维张量，分别在每一深度上进行二维张量的加法运算，示例结果如图 2-37 所示。

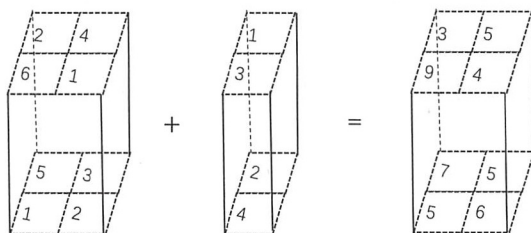


图 2-37 x 与 2 行 1 列 2 深度的三维张量相加

同理， x 与 1 行 2 列 2 深度的三维张量 y 相加，结果如图 2-38 所示。

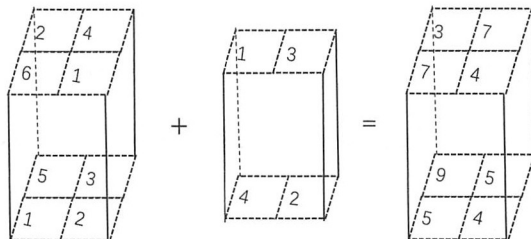


图 2-38 x 与 1 行 2 列 2 深度的三维张量相加

x 还可以与 1 行 1 列 2 深度的三维张量 y 求和，即在每一个深度上分别与 y 对应的深度上的值相加，结果如图 2-39 所示。

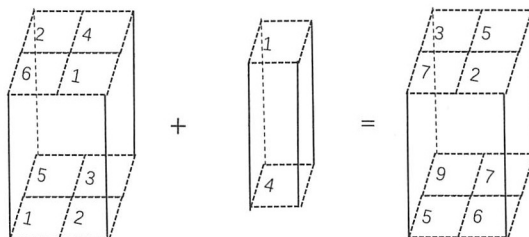
图 2-39 x 与 1 行 1 列 2 深度的三维张量相加

图 2-39 对应的代码，只需修改图 2-36 对应的代码中的 y 值，即

```
y=tf.constant([[[1,4]]],tf.float32)
```

在以上示例的代码实现中，我们也可以简化 y 的写法，将 y 改为一个 1 行 2 列的二维张量，即

```
y=tf.constant([[1,4]],tf.float32)
```

x 可以与一个 1 行 2 列的二维张量相加。对于该示例，我们可以直接将 y 简化为一个长度是 2 的一维张量，即

```
y=tf.constant([1,4],tf.float32)
```

我们将以上内容总结如下：假设有 H 行 W 列 D 深度的三维张量，我们分别为每一深度加一个不同的值，就需要 D 个值，在利用函数 `add` 实现该情况时，可以把 D 个值存储在一个 1 行 1 列 D 深度的三维张量中，也可以把这 D 个值存储在一个 1 行 D 列的二维张量中，最简单的方法是把这 D 个值存储在一个长度是 D 的二维张量中，这种用法在后续章节中会经常用到。

将方法推广到四维张量的加法运算： N 个 H 行 W 列 D 深度的三维张量可以与长度为 D 的一维张量相加，即 N 个三维张量分别与一维张量相加，如图 2-40 所示。

图 2-40 所示示例的对应代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
#"四维张量"
x=tf.constant(
    [
        #"第1个2行2列2深度的三维张量"
        [
            [[2,5],[4,3]],
```



```

[[6,1],[1,2]]
],
#"第2个2行2列2深度的三维张量"
[
[[3,9],[5,7]],
[[7,5],[2,6]]
]
],tf.float32
)
#"长度为2的一维张量"
y=tf.constant([10,20],tf.float32)
#"加法运算"
result=tf.add(x,y)
#"创建会话，打印结果"
session=tf.Session()
print(session.run(result))

```

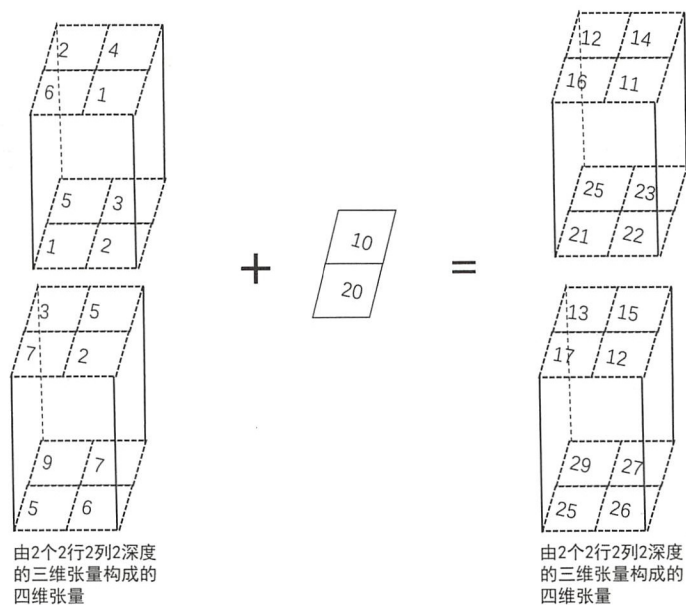


图 2-40 四维张量与一维张量相加

打印结果如下：

```
[
#"第1个三维张量与一维张量的和"
[
[[12. 25.],[14. 23.]],
[[16. 21.],[11. 22.]]
],
#"第2个三维张量与一维张量的和"
[
[[13. 29.],[15. 27.]],
[[17. 25.],[12. 26.]]
]
]
```

根据上面的介绍，加法函数 `add` 可以实现很多不同的要求。TensorFlow 提供的减法（-）函数 `subtract`、乘法（*）函数 `multiply`、除法（/）函数 `divide` 的实现与加法（+）运算类似，我们在此不再赘述。

2.4.2 乘法

TensorFlow 除了提供乘法函数 `multiply`，还提供关于矩阵（二维张量）乘法的函数 `matmul`，示例如下：

$$y = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} -1 \\ -2 \end{bmatrix} = \begin{bmatrix} -5 \\ -11 \end{bmatrix}$$

以上示例的代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
#"2行2列的矩阵"
x=tf.constant(
    [[1,2],[3,4]]
    ,tf.float32
)
#"2行1列的矩阵"
w=tf.constant([[-1],[-2]],tf.float32)
#"矩阵的乘法"
```

```

y=tf.matmul(x,w)
#"创建会话"
session=tf.Session()
#"打印矩阵相乘后的结果"
print(session.run(y))

```

打印结果如下：

```

[[ -5.]
 [-11.]]

```

接下来介绍另一类多个张量的运算：张量的连接。

2.4.3 张量的连接

对于张量的连接，我们还是从简单的二维张量的连接开始，然后推广到更高维张量的连接。

1. 二维张量的连接

我们以图 2-41 所示的两个二维张量为例介绍二维张量的连接。图 2-41 所示分别为一个 2 行 3 列的二维张量和一个 2 行 2 列的二维张量，以上两个二维张量沿“1”方向进行连接，其结果如图 2-42 所示。

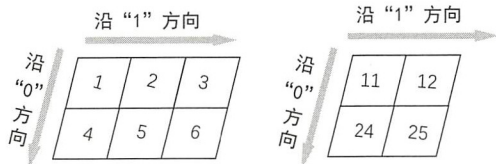


图 2-41 两个二维张量



图 2-42 两个二维张量沿“1”方向进行连接

TensorFlow 通过函数 `concat` 实现张量的连接，以上示例对应的代码如下：

```

import tensorflow as tf
t1=tf.constant(
    [
        [1,2,3],
        [4,5,6]
    ]
)

```

```

    ],tf.float32
)
t2=tf.constant(
    [
    [11,12],
    [24,25]
    ],tf.float32
)
#"两个张量沿"1"方向连接"
t=tf.concat([t1,t2],axis=1)
session=tf.Session()
print(session.run(t))

```

显然，如果二维张量可以沿一个方向进行连接，那么在另一个方向上的维数一定是相等的。如上述示例，沿“1”方向连接，那么这两个张量在“0”方向上的维数一定相等，且为2，即行数为2。相反，这2个张量沿“0”方向是不能进行连接的，因为列数不相等。

2. 三维张量的连接

我们以图 2-43 所示的两个三维张量为例，介绍其在不同方向的连接。两者沿“0”方向连接的结果如图 2-44 所示。

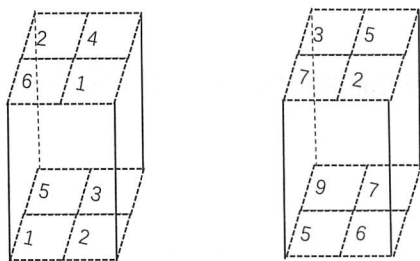


图 2-43 两个三维张量

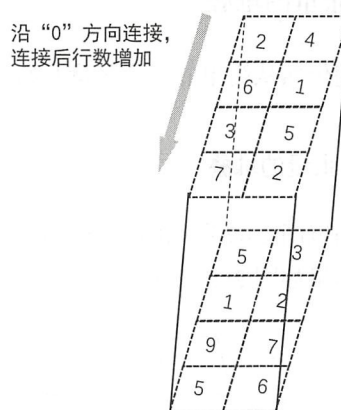


图 2-44 两者沿“0”方向连接

同理，两者沿“1”方向连接的结果如图 2-45 所示，两者沿“2”方向连接的结果如图 2-46 所示。

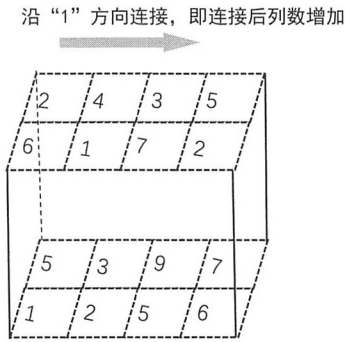


图 2-45 两者沿“1”方向连接

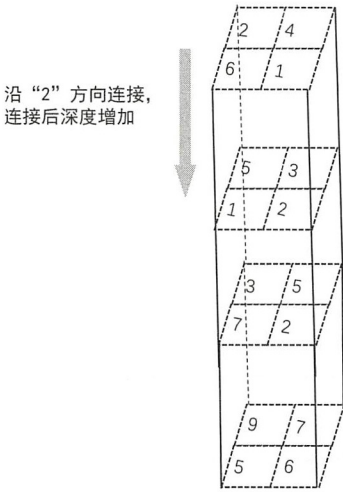


图 2-46 两者沿“2”方向连接

显然，关于三维张量的连接，如果在“0”方向上连接，则列数、深度是相等的。在“1”方向上连接，则行数、深度是相等的；在“2”方向上连接，则行数、列数是相等的，即两个三维张量在一个方向能够连接，那么这两个张量在另两个方向上的维数是相等的。

张量的连接并没有改变张量的维数，如二维张量的连接结果仍是二维张量。以下介绍的张量的堆叠，同连接的操作类似，但其结果却增加了张量的维数。

2.4.4 张量的堆叠

我们仍然从简单的一维张量开始介绍张量的堆叠。

1. 一维张量的堆叠

我们以图 2-47 所示的两个一维张量为例介绍一维张量的堆叠，介绍其沿“0”方向的堆叠，其结果如图 2-48 所示。

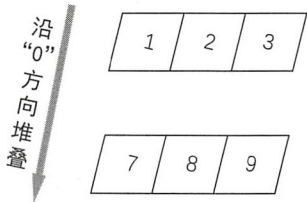


图 2-47 两个一维张量沿“0”方向堆叠

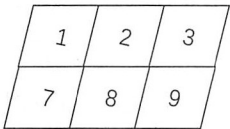


图 2-48 沿“0”方向堆叠的结果

TensorFlow 通过函数 `stack` 实现张量的堆叠，以上示例对应的代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
#"长度为3的一维张量"
t1=tf.constant([1,2,3],tf.float32)
#"长度为3的一维张量"
t2=tf.constant([7,8,9],tf.float32)
#"在"0"方向上堆叠"
t=tf.stack([t1,t2],0)
session=tf.Session()
#"打印结果"
print(session.run(t))
```

打印结果如下：

```
[[ 1.  2.  3.]
 [ 7.  8.  9.]]
```

同样，两者沿“1”方向堆叠，如图 2-49 所示，结果如图 2-50 所示。

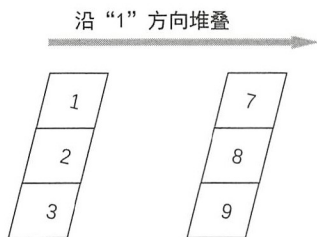


图 2-49 两个一维张量沿“1”方向堆叠

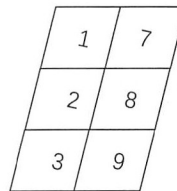


图 2-50 沿“1”方向堆叠的结果

只需要修改函数 `stack` 中的参数即可，对应代码如下：

```
t=tf.stack([t1,t2],1)
```

打印结果如下：

```
[[ 1.  7.]
 [ 2.  8.]
 [ 3.  9.]]
```

接下来介绍二维张量的堆叠。

2. 二维张量的堆叠

我们以图 2-51 所示的两个二维张量为例，介绍二维张量在不同方向的堆叠。首先介绍沿“0”方向的堆叠，结果如图 2-52 所示。

11	12	13
14	15	16

4	5	6
7	8	9

图 2-51 两个二维张量

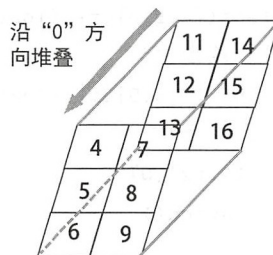


图 2-52 沿“0”方向的堆叠

以上示例对应的代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
t1=tf.constant(
    [
        [11,12,13],
        [14,15,16]
    ],
    tf.float32
)
t2=tf.constant(
    [
        [4,5,6],
        [7,8,9]
    ],tf.float32
)
session=tf.Session()
#"在"0"方向上堆叠"
t=tf.stack([t1,t2],0)
#"打印结果"
print(session.run(t))
```

打印结果如下：

```
[[[ 11.  12.  13.]
   [ 14.  15.  16.]]
```

```
[[ 1.  2.  3.]
 [ 4.  5.  6.]]]
```

同理，两者沿“1”方向堆叠，结果如图 2-53 所示。

对应代码如下：

```
t=tf.stack([t1,t2],1)
```

打印结果如下：

```
[[[ 11.  12.  13.]
   [ 1.  2.  3.]]
```

```
[[ 14.  15.  16.]
 [ 4.  5.  6.]]]
```

两者沿“2”方向堆叠，结果如图 2-54 所示。

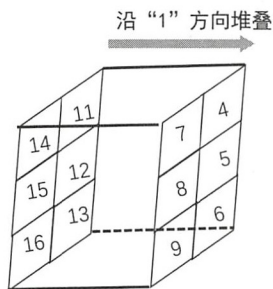


图 2-53 沿“1”方向堆叠

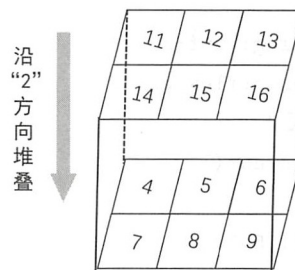


图 2-54 沿“2”方向堆叠

代码如下：

```
t=tf.stack([t1,t2],2)
```

打印结果如下：

```
[[[ 11.  4.]
   [ 12.  5.]
   [ 13.  6.]],
 [[ 14.  7.]
```

```
[ 15.   8.]
[ 16.   9.]]]
```

我们发现堆叠后结果的维数增加了，从两个二维张量变为三维张量。两个尺寸不一样的张量可以进行连接，但是进行堆叠操作的两个张量的尺寸必须相同。

本章最后，介绍多个张量之间的运算：张量的对比。

2.4.5 张量的对比

张量的对比是指两个相同尺寸的张量对应位置的值是否相同，示例如图 2-55 所示。

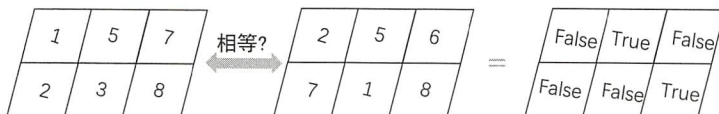


图 2-55 两个张量的对比

若两个张量对应位置的值相等，则返回 `True`；若两个张量对应位置的值不相等，则返回 `False`。TensorFlow 通过函数 `equal` 实现该功能，具体代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
#"二维张量"
t1=tf.constant(
    [
        [1,5,7],
        [2,3,8]
    ],tf.float32
)
#"二维张量"
t2=tf.constant(
    [
        [2,5,6],
        [7,1,8]
    ],tf.float32
)
#"两个张量的对比"
```

```

r=tf.equal(t1,t2)
#"创建会话"
session=tf.Session()
#"打印结果"
print(session.run(r))

```

打印结果如下：

```

[[False  True False]
 [False False  True]]

```

以上只是介绍了常用的一些张量运算的接口，其他的运算接口后面章节中碰到时再详细介绍。下面介绍 TensorFlow 中另一个重要的概念：占位符（placeholder）。

2.5 占位符

占位符的作用及其示例

对占位符的理解，可以将其看作函数的未知数。假设函数 $f(\mathbf{x}) = \mathbf{w}\mathbf{x}$ ，其中 \mathbf{w} 是 3 行 2 列的矩阵，

$$\mathbf{w} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

那么 \mathbf{x} 只要满足行数为 2，列数不定的矩阵就可以了，比如 \mathbf{x} 可以是一个 2 行 2 列的矩阵 $\mathbf{x} = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$ ， \mathbf{x} 也可以是一个 2 行 1 列的矩阵 $\mathbf{x} = \begin{bmatrix} -1 \\ 2 \end{bmatrix}$ ，以上问题，利用占位符更容易表示，示例代码如下：

```

# -*- coding: utf-8 -*-
import tensorflow as tf
import numpy as np
#"占位符"
x=tf.placeholder(tf.float32,[2,None],name='x')
#"3行2列的矩阵"
w=tf.constant(
    [

```

```

        [1,2],
        [3,4],
        [5,6]
    ],tf.float32
)
#"矩阵相乘"
y=tf.matmul(w,x)
#"创建会话"
session=tf.Session()
#"令x为2行2列的矩阵"
result1=session.run(y,feed_dict={x:np.array([[2,1],[1,2]],np.float32)})
print(result1)
#"令x为2行1列的矩阵"
result2=session.run(y,feed_dict={x:np.array([[-1],[2]],np.float32)})
print(result2)

```

打印结果如下：

```

[[ 4.  5.]
 [10. 11.]
 [16. 17.]]
[[3.]
 [5.]
 [7.]]

```

从以上代码可以看出，我们通过会话成员函数 `run` 中的参数 `feed_dict` 给占位符（未知数）赋不同的值。

2.6 Variable 对象

Tensor 对象的值是不可变的，Tensor 类并没有提供任何成员函数改变其值，而且无法用同一个 Tensor 对象记录一个随时变化的值。TensorFlow 中的 Variable 类可以解决该问题，保存随时变化的值。我们创建一个 Variable 对象，将其初始化并打印其值，然后利用其成员函数 `assign_add` 改变其自身的值并打印，具体代码如下：

```

# -*- coding: utf-8 -*-
import tensorflow as tf

```



```

#"创建1个Variable对象"
v=tf.Variable(tf.constant([2,3],tf.float32))
#"创建会话"
session=tf.Session()
#"Variable对象初始化"
session.run(tf.global_variables_initializer())
#"打印值"
print('v初始化的值')
print(session.run(v))
#"利用成员函数assign_add改变本身的值"
session.run(v.assign_add([10,20]))
print('v的当前值')
print(session.run(v))

```

打印结果如下：

```

"v初始化的值"
[2. 3.]
"v的当前值"
[12. 23.]

```

注意：创建 Variable 对象后，要调用方法 `global_variables_initializer()`，才可以使用 Variable 对象的值，否则会报错。

本章介绍了 TensorFlow 的基本数据结构和操作，为后续章节利用 TensorFlow 实现更复杂的问题打下了坚实的基础。第 3 章中，我们需要回忆线性代数中梯度的概念和计算方式，以及涉及的解决机器学习问题中非常重要的模块：梯度下降。

3

梯度及梯度下降法

本章我们主要介绍线性代数中的梯度及其最优化问题中的梯度下降法，这些内容在机器学习中起着至关重要的作用，3.1 节和 3.2 节的内容会显得有些跳跃，但基本是围绕函数的梯度展开的，理解了这两节的内容，可以更好地引出梯度下降法。

3.1 梯度

假设有多元函数

$$F(\mathbf{x}) = F(x_1, x_2, \dots, x_n)$$

用符号 $\frac{\partial F}{\partial \mathbf{x}}$ 代表函数 F 分别对每一个自变量求导，即

$$\frac{\partial F}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial F}{\partial x_1} \\ \frac{\partial F}{\partial x_2} \\ \vdots \\ \frac{\partial F}{\partial x_n} \end{bmatrix}$$

举例：假设二元函数 $F(x_1, x_2) = (3x_1 + 4x_2)^2$ ，计算 F 在点 $(2, 3)$ 处的梯度。 F 针对 x_1 、 x_2 的偏导数如下：

$$\frac{\partial F}{\partial x_1} = 18x_1 + 24x_2, \quad \frac{\partial F}{\partial x_2} = 24x_1 + 32x_2$$

所以 F 在 $(2, 3)$ 处的梯度为 $(18 \times 2 + 24 \times 3, 24 \times 2 + 32 \times 3) = (108, 144)$, 记

$$\frac{\partial F}{\partial \mathbf{x}}|_{\mathbf{x}=(2,3)} = \begin{bmatrix} 108 \\ 144 \end{bmatrix}$$

TensorFlow 通过函数 `gradients(ys, xs, grad_ys=None, name="gradients")` 实现自动计算梯度, 以上示例的具体代码如下。

首先, 将自变量声明为一个占位符:

```
import tensorflow as tf
x=tf.placeholder(tf.float32,(2,1))
```

然后, 利用矩阵乘法函数 `matmul` 实现 $3x_1 + 4x_2$:

```
w=tf.constant([[3,4]],tf.float32)
y=tf.matmul(w,x)
```

接着, 求上述和的平方, 构造出函数 F :

```
F=tf.pow(y,2)
```

利用函数 `gradients` 计算 F 在 $(2, 3)$ 处的梯度:

```
grads=tf.gradients(F,x)
session=tf.Session()
#"打印梯度值"
print(session.run(grads,{x:np.array([[2],[3]])}))
```

打印结果如下:

```
[array([[ 108.],
        [ 144.]], dtype=float32)]
```

显然, 上述代码计算出的结果和手动计算的结果相等。接下来, 我们回忆一个导数计算的链式法则, 这个法则在后续章节中会经常用到。

3.2 导数计算的链式法则

导数计算的链式法则主要包括: 多个函数和的导数和复合函数的导数, 以下依次介绍。

3.2.1 多个函数和的导数

多个函数和的导数的链式法则：多个函数和的导数等于多个函数导数的和，即假设 $F(\mathbf{x}) = \sum_i F_i(\mathbf{x})$ ，则

$$\frac{\partial F}{\partial \mathbf{x}} = \sum_i \frac{\partial F_i}{\partial \mathbf{x}}$$

举例：假设二元函数 $F_1(x_1, x_2) = x_1^2 + 3x_2$ ，二元函数 $F_2(x_1, x_2) = x_1^3 + 2x_2^2$ ，函数 $F(x_1, x_2) = F_1(x_1, x_2) + F_2(x_1, x_2)$ ，则 $\frac{\partial F}{\partial \mathbf{x}} = \frac{\partial F_1}{\partial \mathbf{x}} + \frac{\partial F_2}{\partial \mathbf{x}}$ ，即

$$\frac{\partial F}{\partial x_1} = \frac{\partial F_1}{\partial x_1} + \frac{\partial F_2}{\partial x_1} = 2x_1 + 3x_1^2, \quad \frac{\partial F}{\partial x_2} = \frac{\partial F_1}{\partial x_2} + \frac{\partial F_2}{\partial x_2} = 3 + 4x_2$$

3.2.2 复合函数的导数

复合函数的导数的链式法则：假设复合函数 $f(g(\mathbf{x}))$ ，该函数针对自变量 \mathbf{x} 的导数如下：

$$\frac{\partial f}{\partial \mathbf{x}} = \frac{\partial f}{\partial g} \frac{\partial g}{\partial \mathbf{x}}$$

通过以下示例理解复合函数的导数：假设 $g_1(x_1, x_2) = x_1 + 2x_2$ ， $g_2(x_1, x_2) = 2x_1x_2$ ， $g_3(x_1) = x_1^2$ ，复合函数 $F(g_1, g_2, g_3) = 2g_1 + g_2^2 + g_3$ 。计算 F 对 x_1 的偏导数。因为 F 是关于 g_1 、 g_2 、 g_3 的函数，而 g_1 、 g_2 、 g_3 又是关于 x_1 的函数，所以有

$$\begin{aligned} \frac{\partial F}{\partial x_1} &= \frac{\partial F}{\partial g_1} \frac{\partial g_1}{\partial x_1} + \frac{\partial F}{\partial g_2} \frac{\partial g_2}{\partial x_1} + \frac{\partial F}{\partial g_3} \frac{\partial g_3}{\partial x_1} = 2 \times 1 + 2g_2 \times 2x_2 + 1 \times 2x_1 \\ &= 2 + 2 \times (2x_1x_2) \times 2x_2 + 2x_1 = 2 + 8x_1x_2^2 + 2x_1 \end{aligned}$$

同理，计算 F 对 x_2 的偏导数，虽然 F 是关于 g_1 、 g_2 、 g_3 的函数，但是只有 g_1 、 g_2 是关于 x_2 的函数，所以有

$$\begin{aligned} \frac{\partial F}{\partial x_2} &= \frac{\partial F}{\partial g_1} \frac{\partial g_1}{\partial x_2} + \frac{\partial F}{\partial g_2} \frac{\partial g_2}{\partial x_2} = 2 \times 2 + 2g_2 \times 2x_1 \\ &= 4 + 2 \times (2x_1x_2) \times 2x_1 = 4 + 8x_1^2x_2 \end{aligned}$$

了解了关于导数的计算，接着我们介绍驻点、极值点和鞍点这三者之间的相互关系，以及这三者与梯度之间的关系。我们从最简单的单变量函数（即一元函数）开始介绍。

3.2.3 单变量函数的驻点、极值点、鞍点

我们依次介绍单变量函数的驻点、极小值点、极大值点和鞍点。

1. 驻点

驻点 (stationary point) 是指函数 $f(x)$ 的一阶导数 $\frac{\partial f}{\partial x} = 0$ 的点。例如, 函数 $f(x) = (x-1)^2$ 的驻点在 $x = 1$ 处, 函数 $f(x) = -(x+1)^2$ 的驻点在 $x = -1$ 处, 函数 $f(x) = x^3$ 的驻点在 $x = 0$ 处, 如图 3-1 所示。

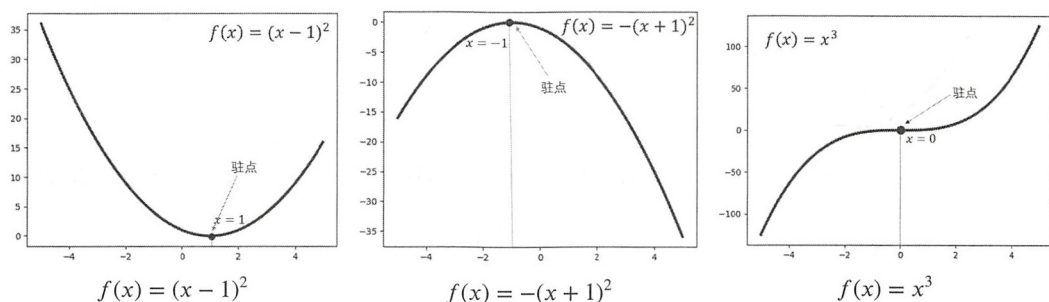


图 3-1 驻点

2. 极小值点

函数在某点的值小于或者等于在该点附近的任何点的函数值, 称为函数的极小值, 该点称为函数的极小值点。

可导函数 $f(x)$ 在 $x = \bar{x}$ 处为极小值的充分必要条件是函数 $f(x)$ 在 $x = \bar{x}$ 处的一阶导数等于 0, 即在 $x = \bar{x}$ 处为驻点, 且 $f(x)$ 在 $x = \bar{x}$ 处的二阶导数 $f''(\bar{x}) > 0$ 。

举例: 函数 $f(x) = (x-1)^2$, 在 $x = 1$ 处是驻点, 又因为 $f(x)$ 在 $x = 1$ 处的二阶导数 $f''(1) = 2 > 0$, 所以 $x = 1$ 是 $f(x)$ 的极小值点, 如图 3-2(a) 所示。

3. 极大值点

与极小值点相反, 若函数在某点的值大于或者等于在该点附近的任何点的函数值, 则称为函数的极大值, 该点称为函数的极大值点。

可导函数 $f(x)$ 在 $x = \bar{x}$ 处为极大值的充分必要条件是函数 $f(x)$ 在 $x = \bar{x}$ 处的一阶导数等于 0, 即在 $x = \bar{x}$ 处为驻点, 且 $f(x)$ 在 $x = \bar{x}$ 处的二阶导数 $f''(\bar{x}) < 0$ 。

举例：函数 $f(x) = -(x + 1)^2$ ， $x = -1$ 处是该函数的驻点，又因为 $f(x)$ 在 $x = -1$ 处的二阶导数 $f''(1) = -2 < 0$ ，所以 $x = -1$ 是 $f(x)$ 的极大值点，如图 3-2(b) 所示。

4. 鞍点

假设函数 $f(x)$ 在 $x = \bar{x}$ 处的一阶导数等于 0，即在 $x = \bar{x}$ 处为驻点，且 $f(x)$ 在 $x = \bar{x}$ 处的二阶导数 $f''(\bar{x}) = 0$ ，则 $f(x)$ 在 $x = \bar{x}$ 处为鞍点，如图 3-2(c) 所示。

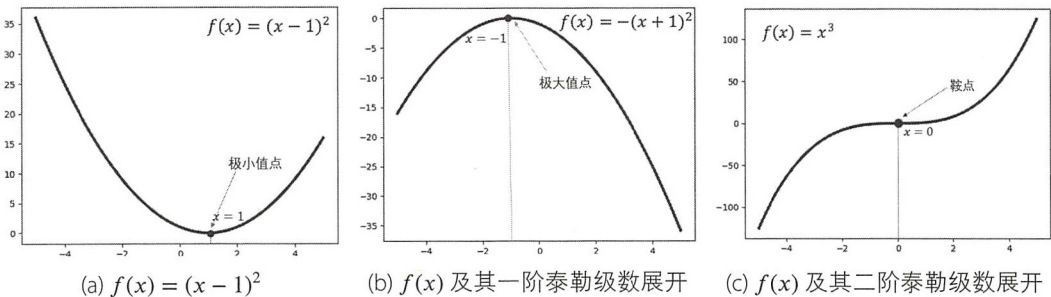


图 3-2 极值点和鞍点

当然，鞍点是不是极值点，要因情况而定。

例如，对于函数 $f(x) = x^4$ ，在 $x = 0$ 处既为驻点也为鞍点，显然 $f(x)$ 在 $x = 0$ 处也为极小值点。函数 $f(x) = -x^4$ 在 $x = 0$ 处既为驻点也为鞍点，显然 $f(x)$ 在 $x = 0$ 处也为极大值点，如图 3-3 所示。

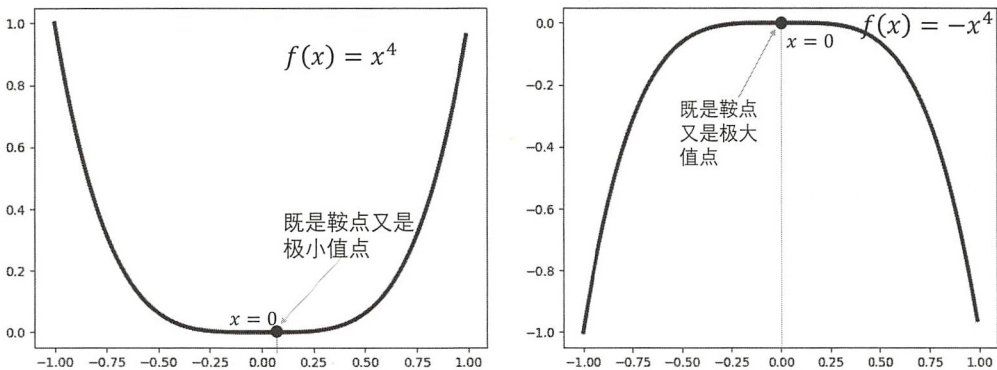


图 3-3 极值点和鞍点的关系

接下来，我们将单变量函数的驻点、极值点、鞍点的概念推广到多变量函数。

3.2.4 多变量函数的驻点、极值点、鞍点

1. 多变量函数的驻点

同单变量函数类似，对于多变量函数 $f(\mathbf{x})$ ， f 关于 \mathbf{x} 的梯度 $\nabla f(\mathbf{x})$ （或记为 $\frac{\partial f}{\partial \mathbf{x}}$ ）等于 0 向量的点称为驻点。

例如，求 $f(x_1, x_2) = 6x_1^2 + 2x_2^2 - 24x_1$ 的驻点，

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 12x_1 - 24 \\ 4x_2 \end{bmatrix} = 0$$

则 $x_1 = 2$ ， $x_2 = 0$ ，即 $f(x_1, x_2)$ 在 $(2, 0)$ 处为驻点。

2. 多变量函数的极值点

将一元可导函数的极值点的充分必要条件推广到多元函数： $\bar{\mathbf{x}}$ 为可导函数 $f(\mathbf{x})$ 的极小值点的充分必要条件是 $\bar{\mathbf{x}}$ 为 $f(\mathbf{x})$ 的驻点，且 $f(\mathbf{x})$ 在 $\bar{\mathbf{x}}$ 处的 Hessian 矩阵 $\nabla^2 f(\bar{\mathbf{x}})$ 的特征值全大于 0。

以上面介绍多变量函数的驻点中的示例为例， $(2, 0)$ 为 $f(x_1, x_2) = 6x_1^2 + 2x_2^2 - 24x_1$ 的驻点，计算在该点处的 Hessian 矩阵：

$$\nabla^2 f(2, 0) = \begin{bmatrix} 12 & 0 \\ 0 & 4 \end{bmatrix}$$

显然，该矩阵的特征值均大于 0，所以 $(2, 0)$ 为 $f(x_1, x_2)$ 的极小值点，如图 3-4 所示。

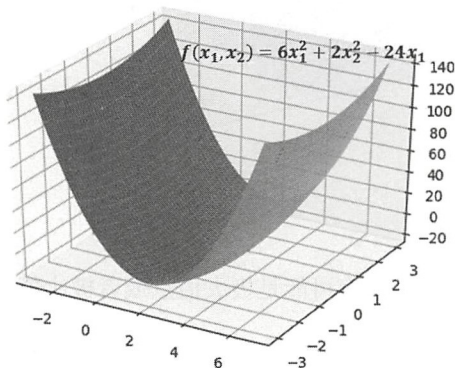


图 3-4 极小值点

相反，如果 Hessian 矩阵的特征值全小于 0，则为极大值点。

3. 多变量函数的鞍点

假设 $\bar{\mathbf{x}}$ 为 $f(\mathbf{x})$ 的驻点，如果 $f(\mathbf{x})$ 在 $\bar{\mathbf{x}}$ 处的 Hessian 矩阵的特征值有的大于 0，有的小于 0，则 $\bar{\mathbf{x}}$ 为 $f(\mathbf{x})$ 的鞍点。

以二元函数 $f(x_1, x_2) = x_1^2 - x_2^2$ 为例。首先，计算该函数的驻点：

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 2x_1 \\ -2x_2 \end{bmatrix} = 0$$

则 $f(x_1, x_2)$ 在 $(0, 0)$ 处为驻点。接着，计算 $f(x_1, x_2)$ 在 $(0, 0)$ 处的 Hessian 矩阵：

$$\nabla^2 f(0, 0) = \begin{bmatrix} 2 & 0 \\ 0 & -2 \end{bmatrix}$$

显然，该矩阵的特征值一个大于 0，一个小于 0，所以 $f(x_1, x_2)$ 在 $(0, 0)$ 处为鞍点，如图 3-5 所示。

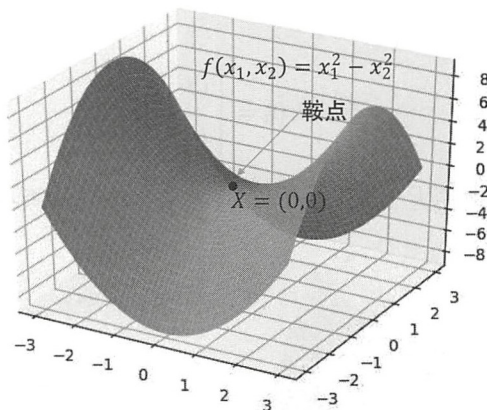


图 3-5 鞍点

同单变量函数类似，鞍点处也有可能是极大值点或者极小值点。

一个函数可能有多个极小值，也被称为局部极小值，这些极小值中最小的被称为最小值或全局最小值，对应的点也被称为最小值点或者全局极小值点，如图 3-6 所示。

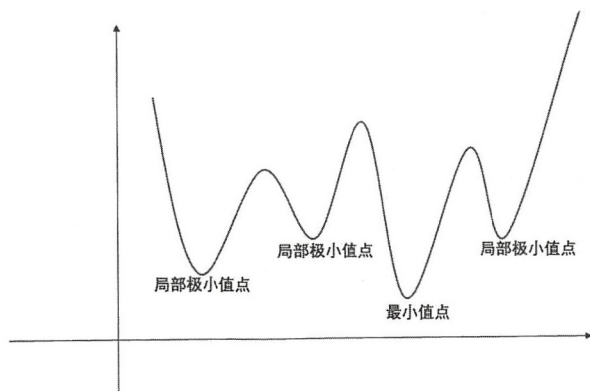


图 3-6 最小值点与局部极小值点

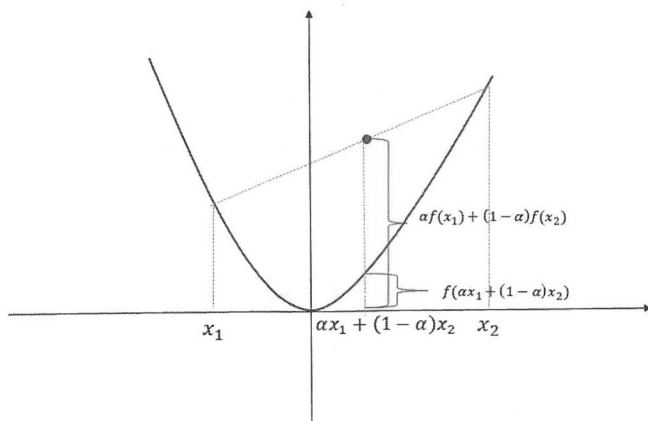
那么存不存在一种函数，使得极小值点和最小值点是同一个点呢？接下来介绍的凸函数就满足这一性质。

4. 凸函数

假设函数 $f(x)$ 满足以下关系：

$$f(\alpha x_1 + (1 - \alpha)x_2) \leq \alpha f(x_1) + (1 - \alpha)f(x_2), \alpha \in [0, 1]$$

则称函数 $f(x)$ 为凸函数。如图 3-7 所示的函数 $f(x) = x^2$ 为凸函数。

图 3-7 凸函数 $f(x) = x^2$

其中凸函数的局部极小值点为全局最小值点（即极小值点也为最小值点）。

以上介绍了函数的驻点、极值点、鞍点及这三者与梯度的关系。接下来，我们介绍函数的泰勒级数展开。

3.2.5 函数的泰勒级数展开

对于函数的泰勒级数展开，我们仍然从简单的一元函数开始，推广到多元函数。

1. 一元函数的泰勒级数展开

一元函数 $f(x)$ 在 $x = x_1$ 处的泰勒展开式为

$$f(x) = f(x_1) + f'(x_1)(x - x_1) + \frac{1}{2!}f''(x_1)(x - x_1)^2 + \dots$$

以函数 $f(x) = x^3 + 2x^2 + 3x + 4$ 为例，如图 3-8(a) 所示，其一阶导数为

$$f'(x) = 3x^2 + 4x + 3$$

二阶导数为

$$f''(x) = 6x + 4$$

三阶导数为

$$f'''(x) = 6$$

所以 $f(x)$ 在 $x = 1$ 处的一阶、二阶、三阶梯度分别为

$$f'(1) = 10, f''(1) = 10, f'''(1) = 6$$

$f(x)$ 在 $x = 1$ 处的一阶泰勒展开式为

$$f_{\text{taylor}_1}(x) = f(1) + f'(1)(x - 1) = 10x$$

如图 3-8(b) 所示。 $f(x)$ 在 $x = 1$ 处的二阶泰勒展开式为

$$f_{\text{taylor}_2}(x) = f(1) + f'(1)(x - 1) + \frac{f''(1)}{2!}(x - 1)^2 = 5x^2 + 5$$

如图 3-8(c) 所示。 $f(x)$ 在 $x = 1$ 处的三阶泰勒展开式为

$$f_{\text{taylor}_3}(x) = f(1) + f'(1)(x - 1) + \frac{f''(1)}{2!}(x - 1)^2 + \frac{f'''(1)}{3!}(x - 1)^3 = x^3 + 2x^2 + 3x + 4$$

显然, $f(x)$ 在 $x = 1$ 处的三阶泰勒展开式 $f_{\text{Taylor}_3}(x)$ 和原函数 $f(x)$ 相等, 从以上结果我们可以看出, $f(x)$ 在 $x = 1$ 处, 越高阶的泰勒级数展开, 越能在 $x = 1$ 处附近更好地逼近 $f(x)$, 到三阶泰勒展开式时, 与原函数完全相等。

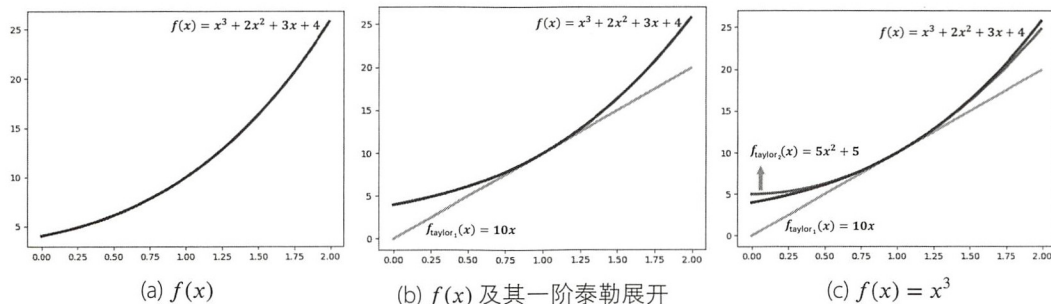


图 3-8 一元函数的泰勒级数展开

接下来, 我们介绍多元函数的泰勒级数展开。

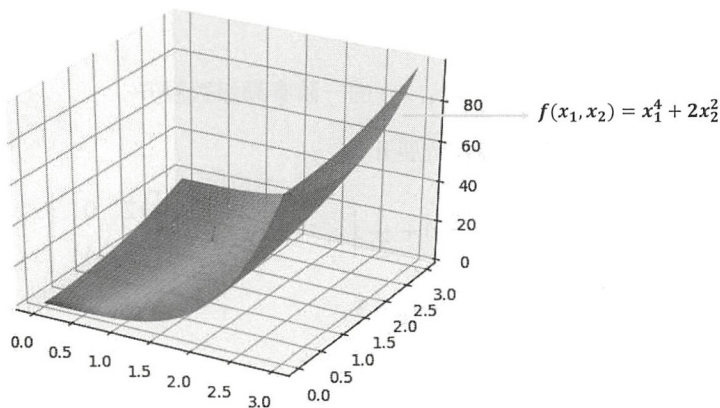
2. 多元函数的泰勒级数展开

假设 N 元函数 $f(\mathbf{x})$, 其中 $\mathbf{x} \in \mathbb{R}^N$, $f(\mathbf{x})$ 在 $\mathbf{x} = \mathbf{x}_1$ 处的泰勒级数展开为

$$f(\mathbf{x}) = f(\mathbf{x}_1) + (\mathbf{x} - \mathbf{x}_1)^T \nabla f(\mathbf{x}_1) + \frac{1}{2}(\mathbf{x} - \mathbf{x}_1)^T \nabla^2 f(\mathbf{x}_1)(\mathbf{x} - \mathbf{x}_1) + \dots$$

其中, $\nabla f(\mathbf{x}_1)$ 代表 $f(\mathbf{x})$ 在 \mathbf{x}_1 处的梯度, $\nabla^2 f(\mathbf{x}_1)$ 代表 $f(\mathbf{x})$ 在 \mathbf{x}_1 处的 Hessian 矩阵。

举例: 假设二元函数 $f(\mathbf{x}) = f(x_1, x_2) = x_1^4 + 2x_2^2$, 如图 3-9 所示。

图 3-9 二元函数 $f(\mathbf{x})$

$f(\mathbf{x})$ 的一阶导数为

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 4x_1^3 \\ 4x_2 \end{bmatrix}$$

$f(\mathbf{x})$ 的 Hessian 矩阵为

$$\nabla^2 f = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} \\ \frac{\partial^2 f}{\partial x_2^2} & \frac{\partial^2 f}{\partial x_2 \partial x_1} \end{bmatrix} = \begin{bmatrix} 12x_1^2 & 0 \\ 0 & 4 \end{bmatrix}$$

则 $f(x_1, x_2)$ 在 $(1, 2)$ 处的一阶泰勒展开为

$$f_{\text{taylor}_1}(x_1, x_2) = f(1, 2) + \begin{bmatrix} x_1 - 1 & x_2 - 2 \end{bmatrix} \begin{bmatrix} 4 \\ 8 \end{bmatrix} = 4x_1 + 8x_2 - 11$$

如图 3-10 所示。

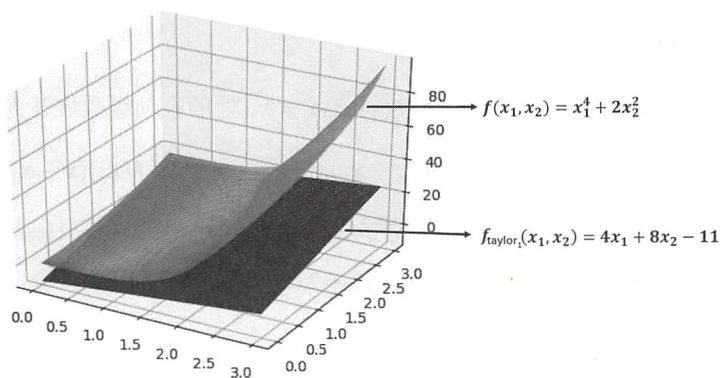


图 3-10 $f(x_1, x_2)$ 的一阶泰勒级数展开

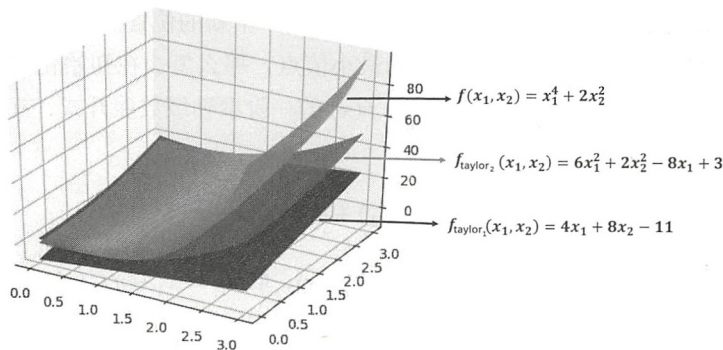
$f(x_1, x_2)$ 在 $(1, 2)$ 处的二阶泰勒级数展开为

$$\begin{aligned} f_{\text{taylor}_2}(x_1, x_2) &= f_{\text{taylor}_1}(x_1, x_2) + \frac{1}{2!} \begin{bmatrix} x_1 - 1 & x_2 - 2 \end{bmatrix} \begin{bmatrix} 12 & 0 \\ 0 & 4 \end{bmatrix} \begin{bmatrix} x_1 - 1 \\ x_2 - 2 \end{bmatrix} \\ &= 6x_1^2 + 2x_2^2 - 8x_1 + 3 \end{aligned}$$

如图 3-11 所示。

同一元函数的泰勒级数展开类似， $f(\mathbf{x})$ 在 x_1 处越高阶的泰勒级数展开，在 x_1 附近能更好地逼近原函数 $f(\mathbf{x})$ 。

了解了以上内容，接下来，我们展开本章，也是贯穿本书的重要内容：梯度下降法。

图 3-11 $f(x_1, x_2)$ 的二阶泰勒级数展开

3.2.6 梯度下降法

我们从简单的一元函数引出梯度下降法。

1. 一元函数的梯度下降法

我们先讨论以下无约束的最小值的最优化问题：

$$\min_x (x - 1)^2$$

即计算函数 $f(x) = (x - 1)^2$ 的最小值点，其中 $x \in \mathbb{R}$ ，因为该函数是凸函数，所以其最小值点即极小值点，根据计算极小值点的充分必要条件，分两步计算该函数的极小值点。

第 1 步，该函数的一阶导数为 $f'(x) = 2(x - 1)$ ，解方程 $f'(x) = 0$ ，显然该方程的解为 $x = 1$ ，即在 $x = 1$ 处为驻点。

第 2 步，该函数的二阶导数为 $f''(x) = 2$ ，显然 $f''(1) = 2 > 0$ ，所以 $x = 1$ 处为 $f(x)$ 的极小值点和最小值点。

上述示例中，方程 $f'(x) = 0$ 是一个一元一次方程，可以非常容易得到该方程的解，也就可以得到 $f(x)$ 的驻点。如果一个函数的一阶导数 $f'(x)$ 是一个一元二次方程，我们也可以得到该方程的解。一元二次方程 $ax^2 + bx + c = 0$ 的解析解为 $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$ ，所谓的解析解就是公式解，但是 5 次及更高次的代数方程的解，不像一元二次代数方程这样具备解析解，也就无法计算出 $f(x)$ 的驻点，找不到其最小值点。

我们换个思路求解以上最优化问题。我们知道计算机是比较擅长按照某种特定的规则进行迭代运算的，于是我们就定义求解凸函数最小值问题的规则。先初始化一个位置点 x_1 ，然后以 x_1 为基础，寻找下一个点 x_2 ，使得 $f(x_2) \leq f(x_1)$ 。以 x_2 为基础，寻找下一个点 x_3 ，使

得 $f(x_3) \leq f(x_2)$ ，依此类推，因为每一次迭代会找到一个新的位置点，该位置点处的函数值比上一次位置点的函数值小，所以总会找到最小值点，那么怎么在初始化 x_1 后，找到一个 x_2 ，使得 $f(x_2) \leq f(x_1)$ 呢？这时就需要用到 $f(x)$ 的一阶泰勒级数展开式，首先将 $f(x)$ 在 x_1 处一阶泰勒级数展开为

$$f_{\text{taylor}_1}(x) = f(x_1) + f'(x_1)(x - x_1)$$

当 x 在 x_1 附近很小的邻域时， $f(x) \approx f_{\text{taylor}_1}(x)$ ，取 $x_2 = x_1 - \eta f'(x_1)$ ，令 η 等于一个很小的大于 0 的数，就可以保证 x_1 在 x_1 附近很小的邻域内，所以有

$$f(x_2) \approx f_{\text{taylor}_1}(x_2) = f(x_1) + f'(x_1)(x_2 - x_1)$$

因为

$$f'(x_1)(x_2 - x_1) = f'(x_1)(x_1 - \eta f'(x_1) - x_1) = -\eta(f'(x_1))^2 \leq 0$$

所以

$$f(x_2) \leq f(x_1)$$

然后，将 $f(x)$ 在 x_2 处一阶泰勒级数展开为

$$f_{\text{taylor}_1}(x) = f(x_2) + f'(x_2)(x - x_2)$$

在 x_2 附近找到 $x_3 = x_2 - \eta f'(x_2)$ ，则

$$f(x_3) \approx f_{\text{taylor}_1}(x_3) = f(x_2) + f'(x_2)(x_3 - x_2)$$

因为

$$f'(x_2)(x_3 - x_2) = f'(x_2)(x_2 - \eta f'(x_2) - x_2) = -\eta(f'(x_2))^2 \leq 0$$

所以

$$f(x_3) \leq f(x_2)$$

依此类推：找到 $x_{n+1} = x_n - \eta f'(x_n)$ ，使 $f(x_{n+1}) \leq f(x_n)$ ，那么什么时候停止寻找呢？只要满足条件 $f(x_{n+1}) - f(x_n) \approx -\eta(f'(x_n))^2 < \epsilon$ 即可，其中 ϵ 是一个很小的值。事实上，满足该条件代表 $(f'(x_n))^2$ 是一个非常小的值，即 $f'(x_n) \approx 0$ ，所以通过迭代寻找的方式，找到了一阶导数等于 0 的点，而对于凸函数来说，一阶导数等于 0 的点为最小值点，该方法就是梯度下降法，是求解凸函数无约束最优化问题常用的方法，其中 η 常称为学习率。

以下就使用这种方法求 $f(x) = (x - 1)^2$ 的最小值点。

初始化:

$$x_1 = 4, \eta = 0.25$$

第 1 次迭代:

$$x_2 = x_1 - \eta f'(x_1) = 4 - 0.25 \times 2 \times (4 - 1) = 2.5$$

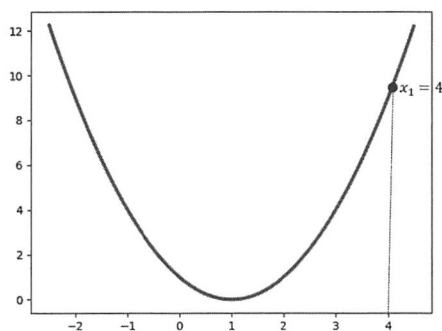
第 2 次迭代:

$$x_3 = x_2 - \eta f'(x_2) = 2.5 - 0.25 \times 2 \times (2.5 - 1) = 1.75$$

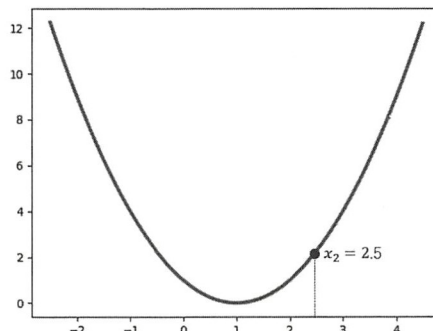
第 3 次迭代:

$$x_4 = x_3 - \eta f'(x_3) = 1.75 - 0.25 \times 2 \times (1.75 - 1) = 1.375$$

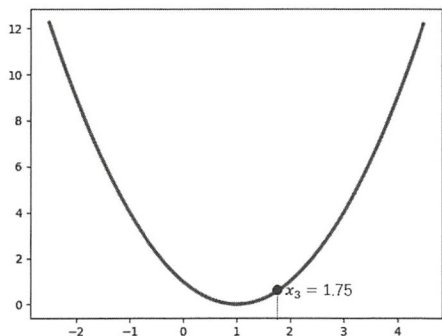
从图 3-12 中可以看出, 随着迭代次数增多, 找到的设置点越来越靠近最小值点。



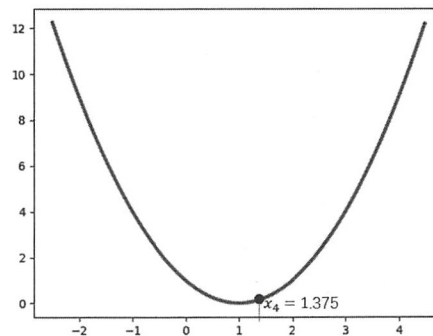
(a) 初始化



(b) 第 1 次迭代



(c) 第 2 次迭代



(d) 第 3 次迭代

图 3-12 一元函数的梯度下降法

TensorFlow 通过函数 `tf.train.GradientDescentOptimizer(learning_rate, use_locking`

=False,name="GradientDescent") 实现以上梯度下降法，利用该函数实现以上示例的迭代过程，具体代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
#"将变量初始化:梯度下降的初始点"
x=tf.Variable(4.0,dtype=tf.float32)
#"函数"
y=tf.pow(x-1,2.0)
#"梯度下降，学习率设置为0.25"
opti=tf.train.GradientDescentOptimizer(0.25).minimize(y)
#"创建会话"
session=tf.Session()
session.run(tf.global_variables_initializer())
#"三次迭代"
for i in range(3):
    session.run(opti)
    #"打印每次迭代的值"
    print(session.run(x))
```

打印结果如下：

```
2.5
1.75
1.375
```

可以看出，打印结果与手动运算的结果相同，其中学习率 $\eta = 0.25$ 设置得比较大，最好设置得小一点，否则会在最小值点附近震荡，稍微修改以上程序，令学习率 $\eta = 0.05$ ，迭代 100 次，每 10 次打印当前寻找的点，并用 matplotlib.pyplot 画出这些点，具体代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
import math
#"将变量初始化:梯度下降的初始点"
x=tf.Variable(15.0,dtype=tf.float32)
#"函数"
y=tf.pow(x-1,2.0)
```

```

#"梯度下降，学习率设置为0.25"
opti=tf.train.GradientDescentOptimizer(0.05).minimize(y)
#"画曲线"
value=np.arange(-15,17,0.01)
y_value=np.power(value-1,2.0)
plt.plot(value,y_value)
#"创建会话"
session=tf.Session()
session.run(tf.global_variables_initializer())
#"三次迭代"
for i in range(100):
    session.run(opti)
    if(i%10==0):
        v=session.run(x)
        plt.plot(v,math.pow(v-1,2.0),'go')
        print('第%d次的x的迭代值: %f'%(i+1,v))
plt.show()

```

输出结果如下：

```

第 1 次的 x 的迭代值: 13.600000
第 11 次的 x 的迭代值: 5.393349
第 21 次的 x 的迭代值: 2.531866
第 31 次的 x 的迭代值: 1.534129
第 41 次的 x 的迭代值: 1.186239
第 51 次的 x 的迭代值: 1.064938
第 61 次的 x 的迭代值: 1.022642
第 71 次的 x 的迭代值: 1.007895
第 81 次的 x 的迭代值: 1.002753
第 91 次的 x 的迭代值: 1.000960

```

从输出结果可以看出，随着迭代次数的增加，找到的位置点越来越靠近最小值点 $x = 1$ 。接下来，我们将以上梯度下降法推广到处理多元函数的最小值问题中。

2. 多元函数的梯度下降法

对于多元函数的梯度下降，同一元函数类似，先讨论无约束的二元凸函数的最优化问题：

$$\min_{x_1, x_2} f(x_1, x_2) = x_1^2 + x_2^2$$

因为该函数是凸函数，所以极小值点为最小值点，根据计算极小值点的充分必要条件，分两步计算该函数的极小值点。

第 1 步，令

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix} = 0$$

即解方程组

$$\begin{cases} 2x_1 = 0 \\ 2x_2 = 0 \end{cases}$$

即 f 在 $(0, 0)$ 处为驻点。

第 2 步，计算 f 在 $(0, 0)$ 点的 Hessian 矩阵：

$$\begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$$

显然，该矩阵的所有特征值均大于 0，所以 f 在 $(0, 0)$ 处为极小值点。因为该函数为凸函数，只有一个极小值点，所以也为最小值点。本示例为一个简单的二元函数，如果该函数是比较复杂的 N 元函数， N 比较大，那么方程组 $\nabla f = 0$ 将变得很难求解，甚至无解析解，所以还是采用迭代寻找的方法，找到 $f(\mathbf{x})$ 的极小值点。

同一元函数寻找极小值点类似，初始化 $\mathbf{x}^{(1)}$ ，然后以 $\mathbf{x}^{(1)}$ 为基础，寻找一个 $\mathbf{x}^{(2)}$ ，使得 $f(\mathbf{x}^{(2)}) \leq f(\mathbf{x}^{(1)})$ 。接着以 $\mathbf{x}^{(2)}$ 为基础，寻找一个 $\mathbf{x}^{(3)}$ ，使得 $f(\mathbf{x}^{(3)}) \leq f(\mathbf{x}^{(2)})$ 。依此类推，寻找到一个 $\mathbf{x}^{(n+1)}$ ，使得 $f(\mathbf{x}^{(n+1)}) \leq f(\mathbf{x}^{(n)})$ 。怎么在初始化 $\mathbf{x}^{(1)}$ 后，找到一个 $\mathbf{x}^{(2)}$ ，使得 $f(\mathbf{x}^{(2)}) \leq f(\mathbf{x}^{(1)})$ 呢？这时就需要用到 $f(\mathbf{x})$ 的一阶泰勒级数展开式，首先将 $f(\mathbf{x})$ 在 $\mathbf{x}^{(1)}$ 处进行一阶泰勒展开：

$$f_{\text{Taylor}_1}(\mathbf{x}) = f(\mathbf{x}^{(1)}) + (\mathbf{x} - \mathbf{x}^{(1)})^T \nabla f(\mathbf{x}^{(1)})$$

当 \mathbf{x} 在 $\mathbf{x}^{(1)}$ 附近很小的邻域内时， $f(\mathbf{x}) \approx f_{\text{Taylor}_1}(\mathbf{x})$ ，那么取

$$\mathbf{x}^{(2)} = \mathbf{x}^{(1)} - \eta \nabla f(\mathbf{x}^{(1)})$$

令 η 等于一个很小的大于 0 的数, 就可以保证 $\mathbf{x}^{(2)}$ 在 $\mathbf{x}^{(1)}$ 附近很小的邻域内, 所以

$$f(\mathbf{x}^{(2)}) \approx f_{\text{Taylor}_1}(\mathbf{x}^{(2)}) = f(\mathbf{x}^{(1)}) + (\mathbf{x}^{(2)} - \mathbf{x}^{(1)})^T \nabla f(\mathbf{x}^{(1)})$$

因为

$$(\mathbf{x}^{(2)} - \mathbf{x}^{(1)})^T \nabla f(\mathbf{x}^{(1)}) = (-\eta \nabla f(\mathbf{x}^{(1)}))^T \nabla f(\mathbf{x}^{(1)}) = -\eta (\|\nabla f(\mathbf{x}^{(1)})\|_2)^2 \leq 0$$

所以

$$f(\mathbf{x}^{(2)}) \leq f(\mathbf{x}^{(1)})$$

同理, 在 $\mathbf{x}^{(2)}$ 附近处, 寻找

$$\mathbf{x}^{(3)} = \mathbf{x}^{(2)} - \eta \nabla f(\mathbf{x}^{(2)})$$

依此类推, 在 $\mathbf{x}^{(n)}$ 附近处, 寻找

$$\mathbf{x}^{(n+1)} = \mathbf{x}^{(n)} - \eta \nabla f(\mathbf{x}^{(n)})$$

什么时候停止寻找呢? 只要

$$f(\mathbf{x}^{(n+1)}) - f(\mathbf{x}^{(n)}) < \epsilon$$

其中 ϵ 代表一个很小的数, 相当于这时 $\nabla f(\mathbf{x}^{(n)}) \approx 0$ 。

接下来, 我们利用梯度下降法寻找上述问题的极小值点。首先, 初始化 $\mathbf{x}^{(1)} = (-4, 4)$, 如图 3-13 所示。

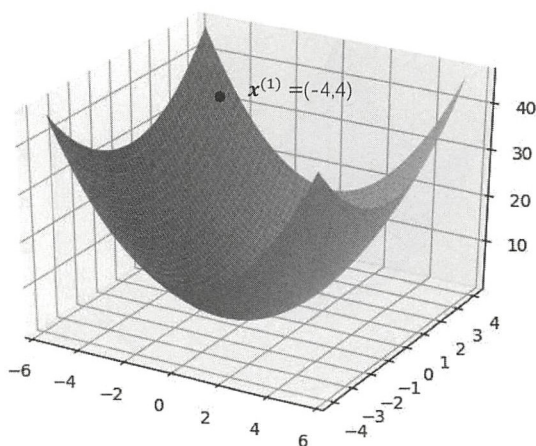


图 3-13 初始化

第 1 次迭代：

$$\mathbf{x}^{(2)} = \mathbf{x}^{(1)} - \eta \nabla f(\mathbf{x}^{(1)}) = \begin{bmatrix} -4 \\ 4 \end{bmatrix} - \frac{1}{4} \times \begin{bmatrix} -8 \\ 8 \end{bmatrix} = \begin{bmatrix} -2 \\ 2 \end{bmatrix}$$

结果如图 3-14 所示。

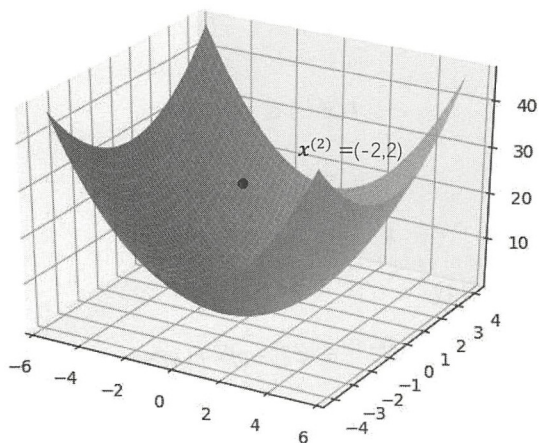


图 3-14 第 1 次梯度下降

第 2 次迭代：

$$\mathbf{x}^{(3)} = \mathbf{x}^{(2)} - \eta \nabla f(\mathbf{x}^{(2)}) = \begin{bmatrix} -2 \\ 2 \end{bmatrix} - \frac{1}{4} \times \begin{bmatrix} -4 \\ 4 \end{bmatrix} = \begin{bmatrix} -1 \\ 1 \end{bmatrix}$$

结果如图 3-15 所示。

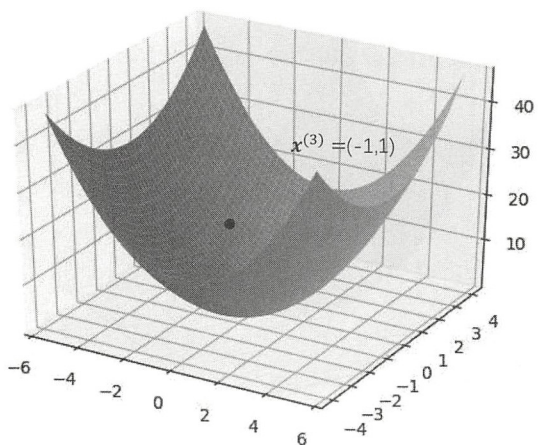


图 3-15 第 2 次梯度下降

依此类推，上述过程的 TensorFlow 实现代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
#"梯度下降的初始点"
x1=tf.Variable(-4.0,dtype=tf.float32)
x2=tf.Variable(4.0,dtype=tf.float32)
#"函数"
y=tf.square(x1)+tf.square(x2)
#"创建会话"
session=tf.Session()
session.run(tf.global_variables_initializer())
#"梯度下降，设置步长为0.25"
opti=tf.train.GradientDescentOptimizer(0.25).minimize(y)
#"第2次迭代"
for i in range(2):
    session.run(opti)
    #"打印每次迭代的值"
    print((session.run(x1),session.run(x2)))
```

打印结果为：

```
(-2.0, 2.0)
(-1.0, 1.0)
```

可以看出，打印结果与手动计算的结果相符，以上代码将二元函数的自变量分开进行初始化，若自变量比较多，这种写法就不方便了，即采用矩阵的形式组织变量，改进后的代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
#"梯度下降的初始点"
x=tf.Variable(tf.constant([-4,4],tf.float32),tf.float32)
#"函数"
y=tf.reduce_sum(tf.square(x))
#"创建会话"
session=tf.Session()
session.run(tf.global_variables_initializer())
#"梯度下降，设置步长为0.25"
```

```
opti=tf.train.GradientDescentOptimizer(0.25).minimize(y)
#"第2次迭代"
for i in range(2):
    session.run(opti)
    #"打印每次迭代的值"
    print(session.run(x))
```

打印结果为：

```
[-2.  2.]
[-1.  1.]
```

综上所述，利用 TensorFlow 对某一函数进行梯度下降，主要分为如下三步。

- (1) 利用 `Variable` 类初始化自变量的值。
- (2) 构造出函数。
- (3) 利用函数 `GradientDescentOptimizer` 进行梯度下降法操作。

利用梯度下降法进行迭代，其实最终收敛的点都是一阶导数等于 0 的点，一阶导数等于 0 的点，对于凸优化来说是最小值点。如果函数不是凸函数，而是具备多个局部极小值的函数，初始化不同的位置点，利用梯度下降法可能会收敛到不同的局部极小值点，如图 3-16 所示。

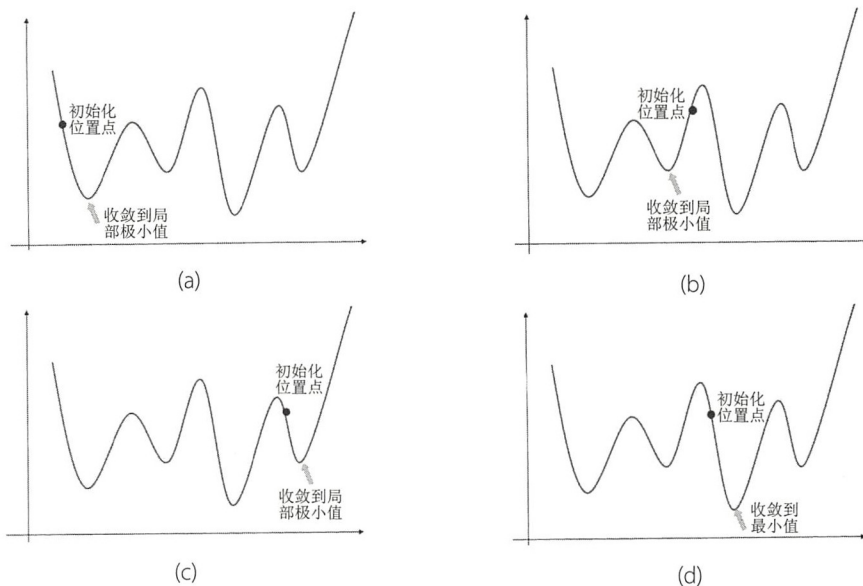


图 3-16 初始化位置点与收敛点的位置关系

如果函数有鞍点，还有可能收敛到鞍点处。以上介绍的是最基本的梯度下降法，下面介绍对其的改进方法，这些方法是为了防止迭代过程中陷入鞍点处或者局部极小值处，但是目前还没有一个完美的解决办法。用每一种梯度下降法单纯地分析公式都不好理解，所以接下来我们针对每一种梯度下降法，举一个简单的例子以便理解。

3.3 梯度下降法

针对标准的梯度下降法，有非常多的改进方法，以下列举一些经典的常用方法，这些方法在 TensorFlow 中也有了相应的实现。

3.3.1 Adagrad 法

假设 N 元函数 $f(\mathbf{x})$ ，其中 $\mathbf{x} = (x_1, x_2, \dots, x_N) \in \mathbb{R}^N$ 。Adagrad^[1] 梯度下降的迭代规则如下：针对每一个自变量，即对 $i = 1, 2, \dots, N$ ，有

$$x_i^{(n+1)} = x_i^{(n)} - \frac{\eta}{\sqrt{\sum_{\tau=1}^n \frac{\partial f}{\partial x_i^{(\tau)}} + \epsilon}} \frac{\partial f}{\partial x_i^{(n)}}$$

用矩阵和向量表示为

$$\mathbf{x}^{(n+1)} = \mathbf{x}^{(n)} - \eta (\mathbf{G}^{(n)})^{-1} \frac{\partial f}{\partial \mathbf{x}^{(n)}}$$

其中

$$\mathbf{G}^{(n)} = \begin{bmatrix} \sqrt{\sum_{\tau=1}^n \frac{\partial f}{\partial x_1^{(\tau)}} + \epsilon} & 0 & \dots & 0 \\ 0 & \sqrt{\sum_{\tau=1}^n \frac{\partial f}{\partial x_2^{(\tau)}} + \epsilon} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \sqrt{\sum_{\tau=1}^n \frac{\partial f}{\partial x_N^{(\tau)}} + \epsilon} \end{bmatrix}$$

通过以下示例了解 Adagrad 梯度下降的迭代过程，假设 $f(\mathbf{x}) = f(x_1, x_2) = x_1^2 + 2x_2^2$ ， f 关于 \mathbf{x} 的一阶导数为

$$\frac{\partial f}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 2x_1 \\ 4x_2 \end{bmatrix}$$

初始化：

$$\mathbf{x}^{(1)} = \begin{bmatrix} 4 \\ 3 \end{bmatrix}, \quad \eta = 0.25, \quad \epsilon = 0.1$$

f 在 $\mathbf{x}^{(1)}$ 处的梯度为

$$\frac{\partial f}{\partial \mathbf{x}}|_{\mathbf{x}=\mathbf{x}^{(1)}} = \begin{bmatrix} 8 \\ 12 \end{bmatrix}$$

第 1 次迭代：

$$\mathbf{x}^{(2)} = \mathbf{x}^{(1)} - \eta \begin{bmatrix} \sqrt{8^2 + \epsilon} & 0 \\ 0 & \sqrt{12^2 + \epsilon} \end{bmatrix}^{-1} \frac{\partial f}{\partial \mathbf{x}}|_{\mathbf{x}=\mathbf{x}^{(1)}} = \mathbf{x}^{(1)} - \eta \begin{bmatrix} \sqrt{8^2 + \epsilon} & 0 \\ 0 & \sqrt{12^2 + \epsilon} \end{bmatrix}^{-1} \begin{bmatrix} 8 \\ 12 \end{bmatrix}$$

计算上式可得：

$$\mathbf{x}^{(2)} = \begin{bmatrix} 4 \\ 3 \end{bmatrix} - 0.25 \times \begin{bmatrix} \sqrt{8^2 + 0.1} & 0 \\ 0 & \sqrt{12^2 + 0.1} \end{bmatrix}^{-1} \begin{bmatrix} 8 \\ 12 \end{bmatrix} = \begin{bmatrix} 3.75019508 \\ 2.75008676 \end{bmatrix}$$

第 2 次迭代：计算 $f(\mathbf{x})$ 在 $\mathbf{x} = \mathbf{x}^{(2)}$ 处的梯度。

$$\begin{aligned} \frac{\partial f}{\partial \mathbf{x}}|_{\mathbf{x}=\mathbf{x}^{(2)}} &= \begin{bmatrix} 2 \times 3.75019508 \\ 4 \times 2.75008676 \end{bmatrix} = \begin{bmatrix} 7.50039016 \\ 11.00034704 \end{bmatrix} \\ \mathbf{x}^{(3)} &= \begin{bmatrix} 3.75019508 \\ 2.75008676 \end{bmatrix} - 0.25 \times \\ &\quad \begin{bmatrix} \sqrt{7.50039016^2 + 0.1} & 0 \\ 0 & \sqrt{11.00034704^2 + 0.1} \end{bmatrix}^{-1} \begin{bmatrix} 7.50039016 \\ 11.00034704 \end{bmatrix} \\ &= \begin{bmatrix} 3.57927611 \\ 2.58118457 \end{bmatrix} \end{aligned}$$

TensorFlow 通过函数 `tf.train.AdagradOptimizer.(learning_rate = 0.001, initial_accumulator_value=0.1)` 实现 Adagrad 梯度下降，上述示例的代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
```



```

#"初始化变量x的值"
x=tf.Variable(tf.constant([[4],[3]],tf.float32),dtype=tf.float32)
w=tf.constant([[1,2]],tf.float32)
y=tf.reduce_sum(tf.matmul(w,t.square(x)))
#"Adagrad梯度下降"
opti=tf.train.AdagradOptimizer(0.25,0.1).minimize(y)
session=tf.Session()
init=tf.global_variables_initializer()
session.run(init)
#"打印前三次的迭代结果"
for i in range(3):
    session.run(opti)
    print(session.run(x))

```

打印结果如下:

```

[[ 3.75019503]
 [ 2.75008678]]
[[ 3.57927608]
 [ 2.58118463]]
[[ 3.4426589 ]
 [ 2.44730377]]

```

3.3.2 Momentum 法

假设 N 元函数 $f(\mathbf{x})$, 其中 $\mathbf{x} = (x_1, x_2, \dots, x_N) \in \mathbb{R}^N$ 。Momentum^[2] 梯度下降的规则如下: 对 $i = 1, 2, \dots, N$, 有

$$v_i^{(n+1)} = \alpha v_i^{(n)} - \eta \frac{\partial f}{\partial x_i^{(n)}}, \quad x_i^{(n+1)} = x_i^{(n)} + v_i^{(n+1)}$$

初始化变量: $v_i^{(0)} = 0, x_i^{(0)}, \eta, \alpha$, 其中 α 是接近 1 的值, 一般设为 0.9。

第 1 次迭代:

$$v_i^{(1)} = \alpha v_i^{(0)} - \eta \frac{\partial f}{\partial x_i |_{x_i=x_i^{(0)}}} = -\eta \frac{\partial f}{\partial x_i |_{x_i=x_i^{(0)}}}, \quad x_i^{(1)} = x_i^{(0)} + v_i^{(1)}$$

第 2 次迭代:

$$v_i^{(2)} = \alpha v_i^{(1)} - \eta \frac{\partial f}{\partial x_i |_{x_i=x_i^{(1)}}} = -\alpha \eta \frac{\partial f}{\partial x_i |_{x_i=x_i^{(0)}}} - \eta \frac{\partial f}{\partial x_i |_{x_i=x_i^{(1)}}}, \quad x_i^{(2)} = x_i^{(1)} + v_i^{(2)}$$

第 3 次迭代：

$$v_i^{(3)} = \alpha v_i^{(2)} - \eta \frac{\partial f}{\partial x_i |_{x_i=x_i^{(2)}}} = -\alpha^2 \eta \frac{\partial f}{\partial x_i |_{x_i=x_i^{(0)}}} - \alpha \eta \frac{\partial f}{\partial x_i |_{x_i=x_i^{(1)}}} - \eta \frac{\partial f}{\partial x_i |_{x_i=x_i^{(2)}}}, \quad x_i^{(3)} = x_i^{(2)} + v_i^{(3)}$$

依此类推，第 n 次迭代：

$$v_i^{(n)} = \alpha v_i^{(n-1)} - \eta \frac{\partial f}{\partial x_i |_{x_i=x_i^{(n-1)}}} = -\eta \sum_{j=0}^{n-1} \alpha^{n-j-1} \frac{\partial f}{\partial x_i |_{x_i=x_i^{(j)}}}, \quad x_i^{(n)} = x_i^{(n-1)} + v_i^{(n)}$$

用向量或者矩阵的形式表示上述迭代过程：

$$\mathbf{v}^{(n+1)} = \alpha \mathbf{v}^{(n)} - \eta \frac{\partial f}{\partial \mathbf{x}^{(n)}}, \quad \mathbf{x}^{(n+1)} = \alpha \mathbf{x}^{(n)} + \mathbf{v}^{(n+1)}$$

通过以下示例了解 Momentum 梯度下降的迭代过程，假设

$$f(\mathbf{x}) = f(x_1, x_2) = x_1^2 + 2x_2^2$$

初始化： $\mathbf{x}^{(0)} = (x_1^{(0)}, x_2^{(0)}) = (4, 3)$ ，学习率 $\eta = 0.01$ ， $\alpha = 0.9$ ， $\mathbf{v}^{(0)} = (v_1^{(0)}, v_2^{(0)}) = (0, 0)$

第 1 次迭代：

$$v_1^{(1)} = \alpha v_1^{(0)} - \eta \frac{\partial f}{\partial x_1 |_{x_1=x_1^{(0)}}} = 0 - 0.01 \times (2 \times 4) = -0.08$$

$$x_1^{(1)} = x_1^{(0)} + v_1^{(1)} = 4 - 0.08 = 3.92$$

$$v_2^{(1)} = \alpha v_2^{(0)} - \eta \frac{\partial f}{\partial x_2 |_{x_2=x_2^{(0)}}} = 0 - 0.01 \times 4 \times 3 = -0.12$$

$$x_2^{(1)} = x_2^{(0)} + v_2^{(1)} = 3 - 0.12 = 2.88$$

即第 1 次迭代后的值 $\mathbf{x}^{(1)} = (x_1^{(1)}, x_2^{(1)}) = (3.92, 2.88)$

第 2 次迭代：

$$v_1^{(2)} = \alpha v_1^{(1)} - \eta \frac{\partial f}{\partial x_1 |_{x_1=x_1^{(1)}}} = -0.9 \times 0.08 - 0.01 \times (2 \times 3.92) = -0.1504$$

$$x_1^{(2)} = x_1^{(1)} + v_1^{(2)} = 3.92 - 0.1504 = 3.7696$$

$$v_2^{(2)} = \alpha v_2^{(1)} - \eta \frac{\partial f}{\partial x_2 |_{x_2=x_2^{(1)}}} = -0.9 \times 0.12 - 0.01 \times (4 \times 2.88) = -0.2232$$

$$x_2^{(2)} = x_2^{(1)} + v_2^{(2)} = 2.88 - 0.2232 = 2.6568$$

即第 2 次迭代后的值 $\mathbf{x}^{(2)} = (x_1^{(2)}, x_2^{(2)}) = (3.7696, 2.6568)$

对应的 TensorFlow 实现如下:

```
opti=tf.train.MomentumOptimizer(0.01,0.9).minimize(f)
```

输出结果如下:

"第 1 次的迭代值"

```
[[3.92]
```

```
[2.88]]
```

"第 2 次的迭代值"

```
[[3.7696002]
```

```
[2.6568  ]]
```

可以看出, 打印结果与手动计算的结果相等。

3.3.3 NAG 法

NAG 法^[6] 是对 Momentum 的改进, 迭代过程如下:

$$\mathbf{v}^{(n)} = \alpha \mathbf{v}^{(n-1)} - \eta \frac{\partial f}{\partial \mathbf{x}} \Big|_{\mathbf{x}_i = \mathbf{x}_i^{(n-1)}} \quad \mathbf{x}^{(n)} = \mathbf{x}^{(n-1)} - \alpha \mathbf{v}^{(n-1)} + (1 + \alpha) \alpha \mathbf{v}^{(n)}$$

通过以下示例了解 NAG 梯度下降的迭代过程, 假设

$$f(\mathbf{x}) = f(x_1, x_2) = x_1^2 + 2x_2^2$$

初始化:

$$\mathbf{v}^{(0)} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad \mathbf{x}^{(0)} = \begin{bmatrix} 4 \\ 3 \end{bmatrix}, \quad \alpha = 0.9, \quad \eta = 0.01$$

第 1 次迭代:

$$\begin{aligned} \mathbf{v}^{(1)} &= 0.9 \times \mathbf{v}^{(0)} - 0.01 \times \frac{\partial f}{\partial \mathbf{x}} \Big|_{\mathbf{x} = \mathbf{x}^{(0)}} = -0.01 \times \begin{bmatrix} 2 \times 4 \\ 4 \times 3 \end{bmatrix} = \begin{bmatrix} -0.08 \\ -0.12 \end{bmatrix} \\ \mathbf{x}^{(1)} &= \mathbf{x}^{(0)} - \alpha \mathbf{v}^{(0)} + (1 + \alpha) \alpha \mathbf{v}^{(1)} = \begin{bmatrix} 4 \\ 3 \end{bmatrix} + (1 + 0.9) \times \begin{bmatrix} -0.08 \\ -0.12 \end{bmatrix} = \begin{bmatrix} 3.848 \\ 2.772 \end{bmatrix} \end{aligned}$$

第 2 次迭代:

$$\mathbf{v}^{(2)} = 0.9 \times \mathbf{v}^{(1)} - 0.01 \times \frac{\partial f}{\partial \mathbf{x}}|_{\mathbf{x}=\mathbf{x}^{(1)}} = 0.9 \times \begin{bmatrix} -0.08 \\ -0.12 \end{bmatrix} - 0.01 \times \begin{bmatrix} 2 \times 3.848 \\ 4 \times 2.772 \end{bmatrix} = \begin{bmatrix} -0.14896 \\ -0.21888 \end{bmatrix}$$

$$\begin{aligned} \mathbf{x}^{(2)} &= \mathbf{x}^{(1)} - \alpha \mathbf{v}^{(1)} + (1 + \alpha) \mathbf{v}^{(2)} \\ &= \begin{bmatrix} 3.848 \\ 2.772 \end{bmatrix} - 0.9 \times \begin{bmatrix} -0.08 \\ -0.12 \end{bmatrix} + (1 + 0.9) \times \begin{bmatrix} -0.14896 \\ -0.21888 \end{bmatrix} = \begin{bmatrix} 3.636976 \\ 2.464128 \end{bmatrix} \end{aligned}$$

在 3.3.2 节中我们已经介绍了 TensorFlow 通过函数 `MomentumOptimizer` 实现 Momentum 法，该函数中有一个参数 `use_nesterov`，只要将该参数设置为 `True`，就是 TensorFlow 实现的 NAG 法，示例代码如下：

```
# "Nesterov 梯度下降"
opti=tf.train.MomentumOptimizer(0.01,0.9,use_nesterov=True).minimize(f)
```

打印结果如下：

"第1次的迭代值"

```
[[3.848]
```

```
[2.772]]
```

"第2次的迭代值"

```
[[3.636976]
```

```
[2.464128]]
```

3.3.4 RMSprop 法

RMSprop^[3] 梯度下降的迭代过程如下：

$$\mathbf{g}^{(n)} = \rho \mathbf{g}^{(n-1)} + (1 - \rho) \left(\frac{\partial f}{\partial \mathbf{x}}|_{\mathbf{x}=\mathbf{x}^{(n)}} \right)^2, \quad \mathbf{x}^{(n+1)} = \mathbf{x}^{(n)} - \frac{\eta}{\sqrt{\mathbf{g}^{(n)} + \epsilon}} \frac{\partial f}{\partial \mathbf{x}}|_{\mathbf{x}=\mathbf{x}^{(n)}}$$

其中 ρ 是趋近于 1 的值，一般设置为 0.9。通过以下示例了解 RMSprop 梯度下降的迭代过程，假设

$$f(\mathbf{x}) = f(x_1, x_2) = x_1^2 + 2x_2^2$$

初始化：

$$\mathbf{g}^{(0)} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \quad \mathbf{x}^{(1)} = \begin{bmatrix} 4 \\ 3 \end{bmatrix}, \quad \rho = 0.9, \quad \eta = 0.01, \quad \epsilon = 10^{-10}$$

第 1 次迭代:

$$\begin{aligned} \mathbf{g}^{(1)} &= 0.9 \times \mathbf{g}^{(0)} + 0.1 \times \left(\frac{\partial f}{\partial \mathbf{x}} \Big|_{\mathbf{x}=\mathbf{x}^{(1)}} \right)^2 = 0.9 \times \begin{bmatrix} 1 \\ 1 \end{bmatrix} + 0.1 \times \begin{bmatrix} 2 \times 4 \\ 4 \times 3 \end{bmatrix}^2 = \begin{bmatrix} 7.3 \\ 15.3 \end{bmatrix} \\ \mathbf{x}^{(2)} &= \mathbf{x}^{(0)} - \frac{\eta}{\sqrt{\mathbf{g}^{(1)} + \epsilon}} \frac{\partial f}{\partial \mathbf{x}} \Big|_{\mathbf{x}=\mathbf{x}^{(1)}} = \begin{bmatrix} 4 \\ 3 \end{bmatrix} - \frac{0.01}{\sqrt{\mathbf{g}^{(1)} + \epsilon}} * \begin{bmatrix} 8 \\ 12 \end{bmatrix} \\ &= \begin{bmatrix} 4 \\ 3 \end{bmatrix} - \begin{bmatrix} \frac{0.01}{\sqrt{7.3 + \epsilon}} \\ \frac{0.01}{\sqrt{15.3 + \epsilon}} \end{bmatrix} * \begin{bmatrix} 8 \\ 12 \end{bmatrix} = \begin{bmatrix} 3.9703906 \\ 2.9693214 \end{bmatrix} \end{aligned}$$

第 2 次迭代:

$$\begin{aligned} \mathbf{g}^{(2)} &= 0.9 \times \mathbf{g}^{(1)} + 0.1 \times \left(\frac{\partial f}{\partial \mathbf{x}} \Big|_{\mathbf{x}=\mathbf{x}^{(2)}} \right)^2 = 0.9 \times \begin{bmatrix} 7.3 \\ 15.3 \end{bmatrix} + 0.1 \times \begin{bmatrix} 2 \times 3.9703906 \\ 4 \times 2.9693214 \end{bmatrix}^2 = \begin{bmatrix} 12.875601 \\ 27.876991 \end{bmatrix} \\ \mathbf{x}^{(3)} &= \mathbf{x}^{(1)} - \frac{\eta}{\sqrt{\mathbf{g}^{(2)} + \epsilon}} \frac{\partial f}{\partial \mathbf{x}} \Big|_{\mathbf{x}=\mathbf{x}^{(2)}} = \begin{bmatrix} 3.9703906 \\ 2.9693214 \end{bmatrix} - \frac{0.01}{\sqrt{\mathbf{g}^{(2)} + \epsilon}} * \begin{bmatrix} 2 \times 3.9703906 \\ 4 \times 2.9693214 \end{bmatrix} \\ &= \begin{bmatrix} 3.9703906 \\ 2.9693214 \end{bmatrix} - \begin{bmatrix} \frac{0.01}{\sqrt{12.875601 + \epsilon}} \\ \frac{0.01}{\sqrt{27.876991 + \epsilon}} \end{bmatrix} * \begin{bmatrix} 2 \times 3.9703906 \\ 4 \times 2.9693214 \end{bmatrix} = \begin{bmatrix} 3.9482606 \\ 2.946825 \end{bmatrix} \end{aligned}$$

依此类推。TensorFlow 的实现代码如下:

```
opti=tf.train.RMSPropOptimizer(learning_rate=0.01,decay=0.9,epsilon=1e-10)
```

打印结果如下:

```
"第1次的迭代值"
[[3.9703906]
 [2.9693215]]
"第2次的迭代值"
[[3.9482605]
 [2.946826 ]]
```

显然, 打印结果与手动计算的结果相等。

3.3.5 具备动量的 RMSprop 法

具备动量的 RMSprop 法是 RMSprop 和 Momentum 结合的方法，具体迭代过程如下：

$$\begin{aligned}\tilde{\mathbf{x}}^{(n)} &= \mathbf{x}^{(n)} + \alpha \mathbf{v}^{(n-1)}, \quad \mathbf{g}^{(n)} = \rho \mathbf{g}^{(n-1)} + (1 - \rho) \left(\frac{\partial f}{\partial \mathbf{x}} \Big|_{\mathbf{x}=\tilde{\mathbf{x}}^{(n)}} \right)^2 \\ \mathbf{v}^{(n)} &= \alpha \mathbf{v}^{(n-1)} - \frac{\eta}{\sqrt{\mathbf{g}^{(n)} + \epsilon}} * \frac{\partial f}{\partial \mathbf{x}} \Big|_{\mathbf{x}=\tilde{\mathbf{x}}^{(n)}}, \quad \mathbf{x}^{(n+1)} = \mathbf{x}^{(n)} + \mathbf{v}^{(n)}\end{aligned}$$

其中 ρ 和 α 都是趋近于 1 的值，一般设置为 0.9。

通过以下示例了解具备动量的 RMSprop 梯度下降的迭代过程，假设 $f(\mathbf{x}) = f(x_1, x_2) = x_1^2 + 2x_2^2$ 。

初始化：

$$\mathbf{v}^{(0)} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad \alpha = 0.9, \quad \rho = 0.9, \quad \epsilon = 10^{-8}, \quad \mathbf{g}^{(0)} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

第 1 次迭代：

$$\begin{aligned}\tilde{\mathbf{x}}^{(1)} &= \mathbf{x}^{(1)} + \alpha \mathbf{v}^{(0)} = \begin{bmatrix} 4 \\ 3 \end{bmatrix} + 0.9 \times \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 4 \\ 3 \end{bmatrix} \\ \mathbf{g}^{(1)} &= \rho \mathbf{g}^{(0)} + (1 - \rho) \left(\frac{\partial f}{\partial \mathbf{x}} \Big|_{\mathbf{x}=\tilde{\mathbf{x}}^{(1)}} \right)^2 = 0.9 \times \begin{bmatrix} 1 \\ 1 \end{bmatrix} + 0.1 \times \begin{bmatrix} 2 \times 4 \\ 4 \times 3 \end{bmatrix}^2 = \begin{bmatrix} 7.3 \\ 15.3 \end{bmatrix} \\ \mathbf{v}^{(1)} &= \alpha \mathbf{v}^{(0)} - \frac{\eta}{\sqrt{\mathbf{g}^{(1)} + \epsilon}} * \frac{\partial f}{\partial \mathbf{x}} \Big|_{\mathbf{x}=\tilde{\mathbf{x}}^{(1)}} = -\frac{0.01}{\sqrt{\mathbf{g}^{(1)} + \epsilon}} * \begin{bmatrix} 8 \\ 12 \end{bmatrix} \\ &= -\begin{bmatrix} \frac{0.01}{\sqrt{7.3 + \epsilon}} \\ \frac{0.01}{\sqrt{15.3 + \epsilon}} \end{bmatrix} * \begin{bmatrix} 8 \\ 12 \end{bmatrix} = -\begin{bmatrix} 0.02960932 \\ 0.03067859 \end{bmatrix} \\ \mathbf{x}^{(2)} &= \mathbf{x}^{(1)} + \mathbf{v}^{(1)} = \begin{bmatrix} 4 \\ 3 \end{bmatrix} - \begin{bmatrix} 0.02960932 \\ 0.03067859 \end{bmatrix} = \begin{bmatrix} 3.9703906 \\ 2.9693214 \end{bmatrix}\end{aligned}$$

第 2 次迭代：

$$\tilde{\mathbf{x}}^{(2)} = \mathbf{x}^{(2)} + \alpha \mathbf{v}^{(1)} = \begin{bmatrix} 3.9703906 \\ 2.9693214 \end{bmatrix} - 0.9 \times \begin{bmatrix} 0.02960932 \\ 0.03067859 \end{bmatrix} = \begin{bmatrix} 3.943742212 \\ 2.941710669 \end{bmatrix}$$

$$\begin{aligned} \mathbf{g}^{(2)} &= \rho \mathbf{g}^{(1)} + (1 - \rho) \left(\frac{\partial f}{\partial \mathbf{x}} \Big|_{\mathbf{x}=\tilde{\mathbf{x}}^{(2)}} \right)^2 = 0.9 \times \begin{bmatrix} 7.3 \\ 15.3 \end{bmatrix} + 0.1 \times \begin{bmatrix} 2 \times 3.943742212 \\ 4 \times 2.941710669 \end{bmatrix}^2 \\ &= \begin{bmatrix} 12.79124105 \\ 27.61585865 \end{bmatrix} \end{aligned}$$

$$\begin{aligned} \mathbf{v}^{(2)} &= \alpha \mathbf{v}^{(1)} - \frac{\eta}{\sqrt{\mathbf{g}^{(2)} + \epsilon}} * \frac{\partial f}{\partial \mathbf{x}} \Big|_{\mathbf{x}=\tilde{\mathbf{x}}^{(2)}} = 0.9 \times \begin{bmatrix} -0.02960932 \\ -0.03067859 \end{bmatrix} - \frac{0.01}{\sqrt{\mathbf{g}^{(2)} + \epsilon}} * \begin{bmatrix} 2 \times 3.943742212 \\ 4 \times 2.941710669 \end{bmatrix} \\ &= 0.9 \times \begin{bmatrix} -0.02960932 \\ -0.03067859 \end{bmatrix} - \begin{bmatrix} \frac{0.01}{\sqrt{12.79124105 + \epsilon}} \\ \frac{0.01}{\sqrt{27.61585865 + \epsilon}} \end{bmatrix} * \begin{bmatrix} 2 \times 3.943742212 \\ 4 \times 2.941710669 \end{bmatrix} = \begin{bmatrix} -0.04870212 \\ -0.05000210 \end{bmatrix} \end{aligned}$$

$$\mathbf{x}^{(3)} = \mathbf{x}^{(2)} + \mathbf{v}^{(2)} = \begin{bmatrix} 3.9703906 \\ 2.9693214 \end{bmatrix} + \begin{bmatrix} -0.04870212 \\ -0.05000210 \end{bmatrix} = \begin{bmatrix} 3.921688 \\ 2.919319 \end{bmatrix}$$

TensorFlow 的实现代码如下：

```
tf.train.RMSPropOptimizer(learning_rate=0.01,decay=0.9,
                           momentum=0.9,epsilon=1e-10)
```

打印结果如下：

```
"第1次的迭代值"
[[3.9703906]
 [2.9693215]]
"第2次的迭代值"
[[3.9216123]
 [2.9192154]]
```

3.3.6 Adadelta 法

Adadelta^[4] 梯度下降的迭代过程如下：

$$\begin{aligned} \mathbf{g}^{(n)} &= \rho \mathbf{g}^{(n-1)} + (1 - \rho) \left(\frac{\partial f}{\partial \mathbf{x}} \Big|_{\mathbf{x}=\mathbf{x}^{(n)}} \right)^2, \quad \Delta \mathbf{x}^{(n)} = - \frac{\sqrt{\mathbf{h}^{(n-1)} + \epsilon}}{\sqrt{\mathbf{g}^{(n)} + \epsilon}} \frac{\partial f}{\partial \mathbf{x}} \Big|_{\mathbf{x}=\mathbf{x}^{(n)}} \\ \mathbf{h}^{(n)} &= \rho \mathbf{h}^{(n-1)} + (1 - \rho)(\Delta \mathbf{x}^{(n)})^2, \quad \mathbf{x}^{(n+1)} = \mathbf{x}^{(n)} + \Delta \mathbf{x}^{(n)} \end{aligned}$$

TensorFlow 通过函数：

```
tf.train.AdadeltaOptimizer(learning_rate=0.001, rho=0.95, epsilon=1e-8)
```

实现 Adadelta 梯度下降。

3.3.7 Adam 法

Adam^[5] 梯度下降的迭代规则如下：

$$\begin{aligned} \mathbf{m}^{(n)} &= \beta_1 \mathbf{m}^{(n-1)} + (1 - \beta_1) \left(\frac{\partial f}{\partial \mathbf{x}} \Big|_{\mathbf{x}=\mathbf{x}^{(n)}} \right), \quad \mathbf{v}^{(n)} = \beta_2 \mathbf{v}^{(n-1)} + (1 - \beta_2) \left(\frac{\partial f}{\partial \mathbf{x}} \Big|_{\mathbf{x}=\mathbf{x}^{(n)}} \right)^2 \\ \hat{\mathbf{m}}^{(n)} &= \frac{\mathbf{m}^{(n)}}{1 - \beta_1^n}, \quad \hat{\mathbf{v}}^{(n)} = \frac{\mathbf{v}^{(n)}}{1 - \beta_2^n}, \quad \mathbf{x}^{(n+1)} = \mathbf{x}^{(n)} - \frac{\eta}{\sqrt{\hat{\mathbf{v}}^{(n)} + \epsilon}} \hat{\mathbf{m}}^{(n)} \end{aligned}$$

一般令 $\beta_1 = 0.9, \beta_1 = 0.999, \epsilon = 10^{-8}$ 。

通过以下示例了解 Adam 梯度下降的迭代过程，假设

$$f(\mathbf{x}) = f(x_1, x_2) = x_1^2 + 2x_2^2$$

初始化：

$$\mathbf{x}^{(1)} = \begin{bmatrix} 4 \\ 3 \end{bmatrix}, \quad \mathbf{m}^{(0)} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad \mathbf{v}^{(0)} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad \beta_1 = 0.9, \quad \beta_1 = 0.999, \quad \epsilon = 10^{-8}$$

第 1 次迭代：

$$\begin{aligned} \mathbf{m}^{(1)} &= \beta_1 \mathbf{m}^{(0)} + (1 - \beta_1) \left(\frac{\partial f}{\partial \mathbf{x}} \Big|_{\mathbf{x}=\mathbf{x}^{(1)}} \right) = 0.1 \times \begin{bmatrix} 8 \\ 12 \end{bmatrix} = \begin{bmatrix} 0.8 \\ 1.2 \end{bmatrix} \\ \mathbf{v}^{(1)} &= \beta_2 \mathbf{v}^{(0)} + (1 - \beta_2) \left(\frac{\partial f}{\partial \mathbf{x}} \Big|_{\mathbf{x}=\mathbf{x}^{(1)}} \right)^2 = 0.001 \times \begin{bmatrix} 8 \\ 12 \end{bmatrix}^2 = \begin{bmatrix} 0.064 \\ 0.144 \end{bmatrix} \\ \hat{\mathbf{m}}^{(1)} &= \frac{\mathbf{m}^{(1)}}{1 - \beta_1} = \frac{1}{1 - 0.9} \times \begin{bmatrix} 0.8 \\ 1.2 \end{bmatrix} = \begin{bmatrix} 8 \\ 12 \end{bmatrix} \\ \hat{\mathbf{v}}^{(1)} &= \frac{\mathbf{v}^{(1)}}{1 - \beta_2} = \frac{1}{1 - 0.999} \times \begin{bmatrix} 0.064 \\ 0.144 \end{bmatrix} = \begin{bmatrix} 64 \\ 144 \end{bmatrix} \\ \mathbf{x}^{(2)} &= \mathbf{x}^{(1)} - \frac{\eta}{\sqrt{\hat{\mathbf{v}}^{(1)} + \epsilon}} \hat{\mathbf{m}}^{(1)} = \begin{bmatrix} 4 \\ 3 \end{bmatrix} - 0.001 \times \begin{bmatrix} \frac{1}{\sqrt{64 + 10^{-8}}} \\ \frac{1}{\sqrt{144 + 10^{-8}}} \end{bmatrix} * \begin{bmatrix} 8 \\ 12 \end{bmatrix} = \begin{bmatrix} 3.999 \\ 2.999 \end{bmatrix} \end{aligned}$$



第 2 次迭代:

$$\begin{aligned} \mathbf{m}^{(2)} &= \beta_1 \mathbf{m}^{(1)} + (1 - \beta_1) \left(\frac{\partial f}{\partial \mathbf{x}} \Big|_{\mathbf{x}=\mathbf{x}^{(2)}} \right) = 0.9 \times \begin{bmatrix} 0.8 \\ 1.2 \end{bmatrix} + 0.1 \times \begin{bmatrix} 2 \times 3.999 \\ 4 \times 2.999 \end{bmatrix} = \begin{bmatrix} 1.5198 \\ 2.2796 \end{bmatrix} \\ \mathbf{v}^{(2)} &= \beta_2 \mathbf{v}^{(1)} + (1 - \beta_2) \left(\frac{\partial f}{\partial \mathbf{x}} \Big|_{\mathbf{x}=\mathbf{x}^{(2)}} \right)^2 = 0.999 \times \begin{bmatrix} 0.064 \\ 0.144 \end{bmatrix} + 0.001 \times \begin{bmatrix} 2 \times 3.999 \\ 4 \times 2.999 \end{bmatrix}^2 = \begin{bmatrix} 0.127904 \\ 0.28776 \end{bmatrix} \\ \hat{\mathbf{m}}^{(2)} &= \frac{\mathbf{m}^{(2)}}{1 - \beta_1^2} = \frac{1}{1 - 0.9^2} \times \begin{bmatrix} 1.5198 \\ 2.2796 \end{bmatrix} = \begin{bmatrix} 7.998947 \\ 11.997895 \end{bmatrix} \\ \hat{\mathbf{v}}^{(2)} &= \frac{\mathbf{v}^{(2)}}{1 - \beta_2^2} = \frac{1}{1 - 0.999^2} \times \begin{bmatrix} 0.127904 \\ 0.28776 \end{bmatrix} = \begin{bmatrix} 63.983992 \\ 143.951976 \end{bmatrix} \\ \mathbf{x}^{(3)} &= \mathbf{x}^{(2)} - \frac{\eta}{\sqrt{\hat{\mathbf{v}}^{(2)} + \epsilon}} \hat{\mathbf{m}}^{(2)} = \begin{bmatrix} 3.999 \\ 2.999 \end{bmatrix} - 0.001 \times \begin{bmatrix} \frac{1}{\sqrt{63.983992}} \\ \frac{1}{\sqrt{143.951976}} \end{bmatrix} * \begin{bmatrix} 7.998947 \\ 11.997895 \end{bmatrix} = \begin{bmatrix} 3.998 \\ 2.998 \end{bmatrix} \end{aligned}$$

依此类推。TensorFlow 实现的代码如下:

```
tf.train.AdamOptimizer(learning_rate=0.001,beta1=0.9,
                        beta2=0.999,epsilon=1e-8)
```

打印结果如下:

```
"第1次的迭代值"
[[3.999]
 [2.999]]
"第2次的迭代值"
[[3.9980001]
 [2.9980001]]
```

我们介绍了比较常用的梯度下降的迭代过程,当然还有很多方法的改进,本节就不一一介绍了,参考文献 [7] 中给出了这些方法的比较。通常,Adagrad、Adadelata、Adam 法比标准的梯度下降法的速度快,也有可能更好地逃离鞍点。

以上这些方法都是关于利用一阶梯度的,还有一些关于利用二阶梯度的梯度下降法。虽然精度比一阶的高,但计算量也变得非常大,在深度学习领域不太常用,本节不再赘述。

接下来,我们讨论一种比较特殊的函数的梯度下降,这类函数也是机器学习中常碰到的



图解深度学习与神经网络：从张量到 TensorFlow 实现

一种类型函数，该类函数可以写成多个函数的和，即

$$F(\mathbf{x}) = \sum_{i=1}^M f_i(\mathbf{x})$$

且这 M 个函数的形式是类似的，比如可能都是一元二次函数，或者都是指数函数等。这类函数常用的三种梯度下降迭代法常称为 Batch 梯度下降（BGD）、随机梯度下降（SGD）和 mini-Batch 梯度下降（mini-BGD）。

3.3.8 Batch 梯度下降

根据导数的链式法则：

$$\frac{\partial F}{\partial \mathbf{x}} = \sum_{i=1}^M \frac{\partial f_i}{\partial \mathbf{x}}$$

基本的 Batch 梯度下降规则为

$$\mathbf{x}^{(n+1)} \leftarrow \mathbf{x}^{(n)} - \eta \frac{\partial F}{\partial \mathbf{x}} = \mathbf{x}^{(n)} - \eta \sum_{i=1}^M \frac{\partial f_i}{\partial \mathbf{x}}$$

当然，以上规则可以替换成 Adagrad 法、Momentum 法等。

假设

$$F(x_1, x_2) = \sum_{i=1}^3 f_i(x_1, x_2)$$

其中

$$f_1(x_1, x_2) = 2x_1^2 + 3x_2^2, \quad f_2(x_1, x_2) = 3x_1^2 + 2x_2^2, \quad f_3(x_1, x_2) = 5x_1^2 + 5x_2^2$$

那么这三个函数的导数分别为

$$\frac{\partial f_1}{\partial \mathbf{x}} = \begin{bmatrix} 4x_1 \\ 6x_2 \end{bmatrix}, \quad \frac{\partial f_2}{\partial \mathbf{x}} = \begin{bmatrix} 6x_1 \\ 4x_2 \end{bmatrix}, \quad \frac{\partial f_3}{\partial \mathbf{x}} = \begin{bmatrix} 10x_1 \\ 10x_2 \end{bmatrix}$$

针对函数 $F(x_1, x_2)$ 进行梯度下降处理，初始化为

$$\mathbf{x}^{(0)} = \begin{bmatrix} 4 \\ 8 \end{bmatrix}, \quad \eta = 0.01$$



第 1 次迭代:

$$\mathbf{x}^{(1)} = \mathbf{x}^{(0)} - \eta \sum_{i=1}^3 \frac{\partial f_i}{\partial \mathbf{x}} \Big|_{\mathbf{x}=\mathbf{x}^{(0)}} = \begin{bmatrix} 4 \\ 8 \end{bmatrix} - 0.01 \times \left(\begin{bmatrix} 6 \times 4 \\ 4 \times 8 \end{bmatrix} + \begin{bmatrix} 4 \times 4 \\ 6 \times 8 \end{bmatrix} + \begin{bmatrix} 10 \times 4 \\ 10 \times 8 \end{bmatrix} \right) = \begin{bmatrix} 3.2 \\ 6.4 \end{bmatrix}$$

第 2 次迭代:

$$\mathbf{x}^{(1)} = \mathbf{x}^{(0)} - \eta \sum_{i=1}^3 \frac{\partial f_i}{\partial \mathbf{x}} \Big|_{\mathbf{x}=\mathbf{x}^{(1)}} = \begin{bmatrix} 3.2 \\ 6.4 \end{bmatrix} - 0.01 \times \left(\begin{bmatrix} 6 \times 3.2 \\ 4 \times 6.4 \end{bmatrix} + \begin{bmatrix} 4 \times 3.2 \\ 6 \times 6.4 \end{bmatrix} + \begin{bmatrix} 10 \times 3.2 \\ 10 \times 6.4 \end{bmatrix} \right) = \begin{bmatrix} 2.56 \\ 5.12 \end{bmatrix}$$

依此类推。显然, 如果 M 非常大, 那么每次迭代, 都会计算每一个函数 $f_i(\mathbf{x})$ 的梯度, 计算量变得非常巨大, 以下我们对其进行改进, 即使用随机梯度下降法计算。

3.3.9 随机梯度下降

Batch 梯度下降每次迭代需要计算每个函数的梯度, 随机梯度下降每次迭代随机选择一个函数 $f_i(\mathbf{x})$, 基本的迭代规则为

$$\mathbf{x}^{(n+1)} \leftarrow \mathbf{x}^{(n)} - \eta \frac{\partial f_i}{\partial \mathbf{x}}, i \in \{1, 2, \dots, N\}$$

当然, 可以使用 Adagrad 法、Momentum 法等。

仍以 3.3.8 节的示例函数为例, 其中初始化值一样, 学习率 $\eta = 0.05$ 。

第 1 次迭代取 $i = 2$:

$$\mathbf{x}^{(1)} = \mathbf{x}^{(0)} - \eta \frac{\partial f_2}{\partial \mathbf{x}} \Big|_{\mathbf{x}=\mathbf{x}^{(0)}} = \begin{bmatrix} 4 \\ 8 \end{bmatrix} - 0.05 \times \begin{bmatrix} 6 \times 4 \\ 4 \times 8 \end{bmatrix} = \begin{bmatrix} 2.8 \\ 6.4 \end{bmatrix}$$

第 2 次迭代取 $i = 1$:

$$\mathbf{x}^{(2)} = \mathbf{x}^{(1)} - \eta \frac{\partial f_1}{\partial \mathbf{x}} \Big|_{\mathbf{x}=\mathbf{x}^{(1)}} = \begin{bmatrix} 2.8 \\ 6.4 \end{bmatrix} - 0.05 \times \begin{bmatrix} 4 \times 2.8 \\ 6 \times 6.4 \end{bmatrix} = \begin{bmatrix} 2.24 \\ 4.48 \end{bmatrix}$$

第 3 次迭代取 $i = 3$:

$$\mathbf{x}^{(3)} = \mathbf{x}^{(2)} - \eta \frac{\partial f_3}{\partial \mathbf{x}} \Big|_{\mathbf{x}=\mathbf{x}^{(2)}} = \begin{bmatrix} 2.24 \\ 4.48 \end{bmatrix} - 0.05 \times \begin{bmatrix} 10 \times 2.24 \\ 10 \times 4.48 \end{bmatrix} = \begin{bmatrix} 1.12 \\ 2.24 \end{bmatrix}$$

依此类推。

Batch 梯度下降是每次迭代计算每个函数的梯度, 随机梯度下降是每次迭代随机选择一个函数的梯度, 那么介于两者之间的是每次迭代随机选择多个函数的梯度, 即 mini-Batch 梯度下降。



3.3.10 mini-Batch 梯度下降

基本的 mini-Batch 的梯度下降规则为

$$\mathbf{x}^{(n+1)} \leftarrow \mathbf{x}^{(n)} - \eta \sum_{j=1}^N \frac{\partial f_{i_j}}{\partial \mathbf{x}}$$

其中 $N < M$, $i_j \in \{1, 2, 3, \dots, M\}$ 。同样，可以使用 Adagrad 法、Momentum 法等。

仍以 3.3.8 节的函数为例，每次迭代随机选择两个函数（这两个函数可以是重复的），其中初始化 $\eta = 0.05$ 。

第 1 次迭代取 $i = 2$ 和 $i = 3$ ：

$$\mathbf{x}^{(1)} = \mathbf{x}^{(0)} - \eta \left(\frac{\partial f_2}{\partial \mathbf{x}} \Big|_{\mathbf{x}=\mathbf{x}^{(0)}} + \frac{\partial f_3}{\partial \mathbf{x}} \Big|_{\mathbf{x}=\mathbf{x}^{(0)}} \right) = \begin{bmatrix} 4 \\ 8 \end{bmatrix} - 0.05 \times \left(\begin{bmatrix} 6 \times 4 \\ 4 \times 8 \end{bmatrix} + \begin{bmatrix} 10 \times 4 \\ 10 \times 8 \end{bmatrix} \right) = \begin{bmatrix} 0.8 \\ 2.4 \end{bmatrix}$$

第 2 次迭代取 $i = 1$ 和 $i = 2$ ：

$$\mathbf{x}^{(2)} = \mathbf{x}^{(1)} - \eta \left(\frac{\partial f_1}{\partial \mathbf{x}} \Big|_{\mathbf{x}=\mathbf{x}^{(1)}} + \frac{\partial f_2}{\partial \mathbf{x}} \Big|_{\mathbf{x}=\mathbf{x}^{(1)}} \right) = \begin{bmatrix} 0.8 \\ 2.4 \end{bmatrix} - 0.05 \times \left(\begin{bmatrix} 4 \times 0.8 \\ 6 \times 2.4 \end{bmatrix} + \begin{bmatrix} 6 \times 0.8 \\ 4 \times 2.4 \end{bmatrix} \right) = \begin{bmatrix} 0.4 \\ 1.2 \end{bmatrix}$$

第 3 次迭代取 $i = 1$ 和 $i = 3$ ：

$$\mathbf{x}^{(3)} = \mathbf{x}^{(2)} - \eta \left(\frac{\partial f_1}{\partial \mathbf{x}} \Big|_{\mathbf{x}=\mathbf{x}^{(2)}} + \frac{\partial f_3}{\partial \mathbf{x}} \Big|_{\mathbf{x}=\mathbf{x}^{(2)}} \right) = \begin{bmatrix} 0.4 \\ 1.2 \end{bmatrix} - 0.05 \times \left(\begin{bmatrix} 4 \times 0.4 \\ 6 \times 1.2 \end{bmatrix} + \begin{bmatrix} 10 \times 0.4 \\ 10 \times 1.2 \end{bmatrix} \right) = \begin{bmatrix} 0.12 \\ 0.24 \end{bmatrix}$$

依此类推，其中 mini-Batch 梯度下降是深度学习中最常用的规则。

本章介绍了很多梯度下降法及其对应的 TensorFlow 实现。在第 4 章中，作者将介绍如何使用这些规则解决机器学习中常遇到的回归问题，并介绍通过 TensorFlow 解决该类问题时的常用函数接口。

3.4 参考文献

- [1] John Duchi, Elad Hazan, and Yoram Singer. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. Journal of Machine Learning Research, 12:2121-2159, 2011.
- [2] D.E. Rumelhart, G.E. Hinton, and R.J. Williams, "Learning representations by back-propagating errors," Nature, vol. 323, pp. 533-536, 1986.
- [3] http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf



3 梯度及梯度下降法

- [4] Matthew D. Zeiler. ADADELTA: An Adaptive Learning Rate Method. arXiv preprint arXiv:1212.5701, 2012.
- [5] Diederik P. Kingma and Jimmy Lei Ba. Adam: a Method for Stochastic Optimization. International Conference on Learning Representations, pages 1-13, 2015.
- [6] Santanu Pattanayak. Pro Deep Learning with TensorFlow, A Mathematical Approach to Advanced Artificial Intelligence in Python.
- [7] Sebastian Ruder. An overview of gradient descent optimization algorithms, 2017.



4

回归分析

回归分析是确定两种或两种以上变量间相互依赖的定量关系的一种统计分析方法，通常分为两种情况：线性回归分析和非线性回归分析。

4.1 线性回归分析

线性回归分析中只包括一个自变量和一个因变量，且二者的关系可用一条直线近似表示，这种回归分析称为一元线性回归分析。如果回归分析中包括两个或两个以上的自变量，且因变量和自变量之间是线性关系，则称为多元线性回归分析。作者首先介绍一元线性回归模型，以及如何利用 TensorFlow 解决一元线性回归，然后介绍多元线性回归模型。

4.1.1 一元线性回归

假设已知 xoy 二维平面上 N 个点组成的点集 $\{(x^{(i)}, y^{(i)})\} \in \mathbb{R} \times \mathbb{R}, i = 1, 2, 3, \dots, N$ ，求一条直线 $y = wx + b$ ，使得这些点沿 y 方向到直线的距离的平方和（即损失函数）最小，即

$$\min \sum_{i=1}^N (y^{(i)} - (wx^{(i)} + b))^2$$

其中 $(y^{(i)} - (wx^{(i)} + b))^2$ 代表第 i 个点沿 y 轴方向到直线 $y = wx + b$ 的距离的平方。



我们通过以下例子理解以上问题。已知 xoy 平面上的 6 个点 (1,3)、(2,4)、(3,7)、(4,8)、(5,11) 和 (6,14)，寻找一条直线 $y = wx + b$ ，如图 4-1 所示，使得这些点沿 y 轴方向到该直线的距离的平方和最小。

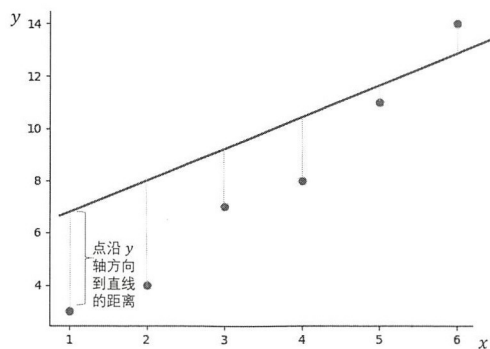


图 4-1 点沿 y 轴方向到该直线的距离

我们可以通过 TensorFlow 求出该直线。因为直线的斜率 w 和截距 b 是要求解的未知量，在对应的代码实现中将这两个未知量分别初始化为两个 `Variable` 对象，而损失函数可以用函数 `square` 和 `reduce_sum` 实现，然后针对构造的损失函数利用梯度下降函数 `GradientDescentOptimizer` 计算 w 和 b ，具体代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
#"6个点的横坐标"
x=tf.constant([1,2,3,4,5,6],tf.float32)
#"6个点的纵坐标"
y=tf.constant([3,4,7,8,11,14],tf.float32)
#"初始化直线的斜率"
w=tf.Variable(1.0,dtype=tf.float32)
#"初始化直线的截距"
b=tf.Variable(1.0,dtype=tf.float32)
#"6个点到直线沿y轴方向距离的平方和"
loss=tf.reduce_sum(tf.square(y-(w*x+b)))
#创建会话
session=tf.Session()
session.run(tf.global_variables_initializer())
```



图解深度学习与神经网络：从张量到 TensorFlow 实现

```

#"梯度下降法"
opti=tf.train.GradientDescentOptimizer(0.005).minimize(loss)
#"记录每一次迭代后的平均平方误差(Mean Squared Error)"
MSE=[]
#"循环500次"
for i in range(500):
    session.run(opti)
    MSE.append(session.run(loss))
    #"每隔50次打印直线的斜率和截距"
    if i%50==0:
        print((session.run(w),session.run(b)))
#"画出损失函数的值"
plt.figure(1)
plt.plot(MSE)
plt.show()
#"画出6个点及最后计算出的直线"
plt.figure(2)
x_array,y_array=session.run([x,y])
plt.plot(x_array,y_array,'o')
xx=np.arange(0,10,0.05)
yy=session.run(w)*xx+session.run(b)
plt.plot(xx,yy)
plt.show()

```

上述代码中，每隔 50 次打印当前 w 和 b 的值，打印结果如下：

```

(1.91, 1.2)
(2.0550959, 0.75369716)
(2.1164339, 0.49109679)
(2.1518073, 0.3396554)
(2.1722074, 0.2523191)
(2.1839721, 0.20195228)
(2.1907568, 0.17290573)
(2.1946695, 0.15615471)
(2.1969259, 0.14649433)
(2.1982272, 0.14092326)

```

记录每次迭代计算的 w 和 b 时，损失函数的值如图 4-2 所示。显然，随着每次迭代，损



失函数的值也在逐渐减小,并最终达到平稳的状态,即得到最后的 w 和 b 的值,并画出该直线,如图 4-3 所示。

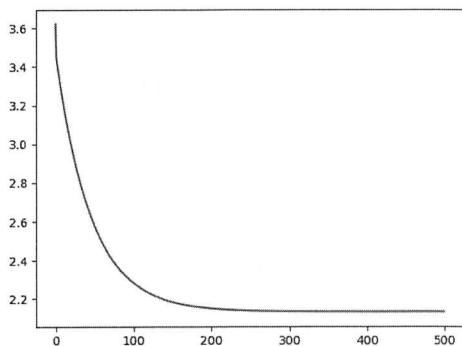


图 4-2 损失函数的值

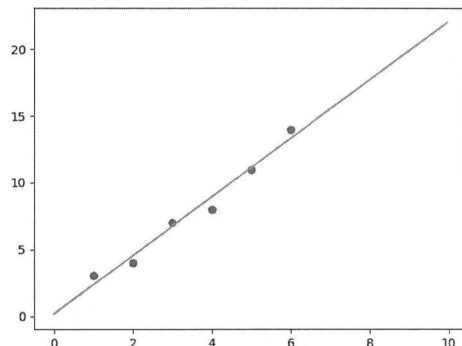


图 4-3 拟合的直线

上述代码中使用的是标准的梯度下降法,对应的函数为 `GradientDescentOptimizer`,也可以使用第 3 章介绍的其他梯度下降法,代码中只需要用函数 `AdagradOptimizer` 代替函数 `GradientDescentOptimizer` 即可。以上代码只实现了计算模型(即 w 和 b 的值)的步骤,并没有对模型进行保存。

为了方便预测时直接使用该模型,需要对其进行保存,接下来介绍如何实现该步骤。

4.1.2 保存和加载回归模型

TensorFlow 保存计算模型,可以简单地理解为用文件保存程序中的 `Variable` 对象。

作者通过以下代码解读该过程。首先,定义两个 `Variable`,并初始化为不同的值。然后,声明一个 `tf.train.Saver` 类的对象,调用该类中的方法 `save`,将这个 `Variable` 对象保存到当前文件夹下的文件名为 `model.ckpt` 的文件中。注意,使用方法 `save` 时,一定要设置文件的保存路径,否则会报错。

```
# -*- coding: utf-8 -*-
import tensorflow as tf

#"第1个Variable, 初始化为一个长度为3的一维张量"
v1=tf.Variable(tf.constant([1,2,3],tf.float32),dtype=tf.float32,name='v1')

#"第2个Variable, 初始化为一个长度为2的一维张量"
v2=tf.Variable(tf.constant([4,5],tf.float32),dtype=tf.float32,name='v2')

#"声明一个tf.train.Saver对象"
saver =tf.train.Saver()
```



图解深度学习与神经网络：从张量到 TensorFlow 实现

```

#"创建会话"
session=tf.Session()
#"初始化变量"
session.run(tf.global_variables_initializer())
#"将变量v1和v2保存到当前文件夹下的model.ckpt文件中"
save_path=saver.save(session,'./model.ckpt')
session.close()

```

运行以上程序，会在当前文件夹下生成文件 `model.ckpt`。

我们接着从该文件中读取其中保存的两个变量的值，变量名与上一段代码相同，且形状也必须相同，作者故意将初始化的值改变了，这样可以分辨出当前 `Variable` 的值是新初始化的，还是从文件中读取的。与保存模型类似，声明一个 `tf.train.Saver` 类的对象，但是调用的不再是方法 `save`，而是对应的该类中的方法 `restore`，具体实现代码如下：

```

# -*- coding: utf-8 -*-
import tensorflow as tf
#"初始化两个变量，变量形状与model.cpkt文件中的必须相等"
v1=tf.Variable([11,12,13],dtype=tf.float32,name='v1')
v2=tf.Variable([15,16],dtype=tf.float32,name='v2')
#"声明一个tf.train.Saver类"
saver =tf.train.Saver()
with tf.Session() as sess:
    #"加载model.ckpt文件"
    saver.restore(sess,'./model.ckpt')
    #"打印两个变量的值"
    print(sess.run(v1))
    print(sess.run(v2))
sess.close()

```

打印结果如下：

```

[1. 2. 3.]
[4. 5.]

```

从打印结果可以看出，虽然该程序将变量 `v1` 初始化为 `[11,12,13]`，但是打印的结果仍然是文件 `/model.ckpt` 中变量 `v1` 的值。

以上方法利用类 `tf.train.Saver` 中的方法 `restore` 读取文件中变量的值，但前提是知道要读取的变量的名称及其形状。以下通过另一种方式，直接获得文件中的变量的名称及其对



应的值，仍以读取上一段程序中提到的文件 `/model.ckpt` 为例，具体代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
from tensorflow.python import pywrap_tensorflow
#ckpt=tf.train.get_checkpoint_state('./')
#"获取在当前文件夹下(./)的ckpt文件，具体根据ckpt保存的位置设置"
ckpt=tf.train.latest_checkpoint('./')
#"打印获取的ckpt文件"
print('获取的ckpt文件:'+ckpt)
#"创建NewCheckpointReader类，读取ckpt文件中的变量名称及其对应的值"
reader=pywrap_tensorflow.NewCheckpointReader(ckpt)
var_to_shape_map=reader.get_variable_to_shape_map()
for key in var_to_shape_map:
    print('tensor_name:',key)
    print(reader.get_tensor(key))
```

打印结果如下：

```
获取的ckpt文件:./model.ckpt
tensor_name: v2
[4. 5.]
tensor_name: v1
[1. 2. 3.]
```

从打印结果可以看出，该程序从文件中获取了变量的名称及其对应的值。

以下程序仍是介绍变量模型的保存和读取这两个过程，目的是熟悉一种常用的用字典类管理 `Variable` 对象的程序书写方式，并熟练掌握保存和读取模型的过程。保存模型的具体代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
#"用字典类管理变量"
weights={
    'w1':tf.Variable([11,12,13],dtype=tf.float32,name='w1'),
    'w2':tf.Variable([21,22],dtype=tf.float32,name='w2')
}
bias={
    'b1':tf.Variable([101,102],dtype=tf.float32,name='b1'),
```



```

        'b2':tf.Variable(2,dtype=tf.float32,name='b2')
    }
    # "创建会话"
    session=tf.Session()
    # "声明一个tf.train.Saver类"
    saver=tf.train.Saver()
    with tf.Session() as sess:
        # "变量初始化"
        sess.run(tf.global_variables_initializer())
        # "将变量保存在当前文件夹下的modelMul.ckpt文件中"
        saver.save(sess,'./modelMul.ckpt')

```

以上程序，将字典类中声明的 4 个变量保存到了当前文件夹下的 modelMul.ckpt 文件中，接着给出读取该文件中这 4 个变量值的代码：

```

# -*- coding: utf-8 -*-
import tensorflow as tf
# "用字典类管理变量"
weights={
    'w1':tf.Variable([1,13,22],dtype=tf.float32,name='w1'),
    'w2':tf.Variable([31,32],dtype=tf.float32,name='w2')
}
bias={
    'b1':tf.Variable([2,12],dtype=tf.float32,name='b1'),
    'b2':tf.Variable(23,dtype=tf.float32,name='b2')
}
# "声明一个tf.train.Saver类"
saver=tf.train.Saver()
with tf.Session() as sess:
    # "加载modelMul.ckpt文件"
    saver.restore(sess,'./modelMul.ckpt')
    # "打印值"
    print(sess.run(weights['w1']))
    print(sess.run(weights['w2']))
    print(sess.run(bias['b1']))
    print(sess.run(bias['b2']))
sess.close()

```

打印结果如下：

```
[11. 12. 13.]
[21. 22.]
[101. 102.]
2.0
```

4.1.3 多元线性回归

我们介绍了如何利用 TensorFlow 处理一元线性回归，并学习掌握了如何用文件保存和读取 Variable 对象。

接下来将一元线性回归推广到多元线性回归，并介绍用 TensorFlow 处理回归问题的完整过程。

(1) 训练多元线性回归模型，并保存模型。

(2) 加载多元回归模型，并进行预测。

假设已知 N 维高维空间中的 K 个点组成的点集 $\{\mathbf{p}^{(i)}\}, \mathbf{p}^{(i)} \in \mathbb{R}^N, i = 1, 2, 3, \dots, K$ ，求该空间中的超平面，使得这些点到该超平面的距离的平方和最小。

我们可以通过简单的三维空间中的例子理解多元线性回归问题。假设 xyz 三维空间中有 6 个点：(1, 1, 8)、(2, 1, 12)、(3, 2, 10)、(1, 2, 14)、(4, 5, 28) 和 (5, 8, 10)，寻找一个超平面

$$z = f(x, y) = w_1 \cdot x + w_2 \cdot y + b$$

使得这些点到超平面（沿 z 轴方向）的距离和（即损失函数）最小。

我们首先介绍回归问题的第 1 个步骤：训练并保存模型。

1. 训练多元线性回归模型，并保存模型

以下代码中，利用梯度下降法进行了 500 次迭代求解该模型，保存了每一次迭代的损失值。每迭代 100 次，利用函数 Print 打印当前模型的值，并将最后计算的模型值保存到当前目录的文件夹 ./regression/下，且文件名为 regressionModel.ckpt，具体实现如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
```

```

#"xyz坐标内的点"
xy=tf.placeholder(tf.float32,[None,2])
z=tf.placeholder(tf.float32,[None,1])
#"初始化 $z=w_1*x+w_2*y+b$ 中的 $w_1$ 、 $w_2$ 、 $b$ "
w=tf.Variable(tf.constant([[1],[1]],tf.float32),dtype=tf.float32)
b=tf.Variable(1.0,dtype=tf.float32)
#"损失函数"
loss=tf.reduce_sum(tf.square(z-(tf.matmul(xy,w)+b)))
#"创建会话"
session=tf.Session()
session.run(tf.global_variables_initializer())
#"梯度下降法"
opti=tf.train.GradientDescentOptimizer(0.005).minimize(loss)
#"记录每一次迭代后的平均平方误差(Mean Squared Error)"
MSE=[]
#"训练数据"
xy_train=np.array([
    [1,1],
    [2,1],
    [3,2],
    [1,2],
    [4,5],
    [5,8]
],np.float32)
z_train=np.array([
    [8],
    [12],
    [10],
    [14],
    [28],
    [10]
],np.float32)
#"声明一个tf.train.Saver类"
saver=tf.train.Saver()
#"训练模型，循环500次"
for i in range(500):

```

```

#"梯度下降"
session.run(opti,feed_dict={xy:xy_train,z:z_train})
#"计算每一次迭代的损失值，并追加到列表中进行保存"
MSE.append(session.run(loss,feed_dict={xy:xy_train,z:z_train}))
#"每隔100次打印w和b的值"
if i%100==0:
    #"保存模型"
    saver.save(session,
                './regression/regressionModel.ckpt',global_step=i)
    print('-----"第"+str(i)+"次的迭代值"-----')
    print(session.run([w,b]))
#"打印第500次(最后)的迭代值"
print('-----"第"'+str(500)+'"次的迭代值"-----')
print(session.run([w,b]))
saver.save(session,'./regression/regressionModel.ckpt',global_step=i)
#"画出损失函数的值"
plt.figure(1)
plt.plot(MSE)
plt.show()

```

打印结果如下：

```

-----"第0次的迭代值"-----
[array([[1.95],
        [1.99]], dtype=float32), 1.41]
-----"第100次的迭代值"-----
[array([[ 2.9388034],
        [-0.6417622]], dtype=float32), 7.5024066]
-----"第200次的迭代值"-----
[array([[ 2.1817584 ],
        [-0.35411417]], dtype=float32), 8.839961]
-----"第300次的迭代值"-----
[array([[ 1.8715947 ],
        [-0.22765213]], dtype=float32), 9.346234]
-----"第400次的迭代值"-----
[array([[ 1.750861 ],
        [-0.1783222]], dtype=float32), 9.542803]

```

-----"第500次的迭代值"-----

```
[array([[ 1.7042251],
        [-0.1592657]]], dtype=float32), 9.618724]
```

从损失值的变化图 4-4 中可以看出，随着迭代次数的增加，损失值逐渐变小并趋于稳定。

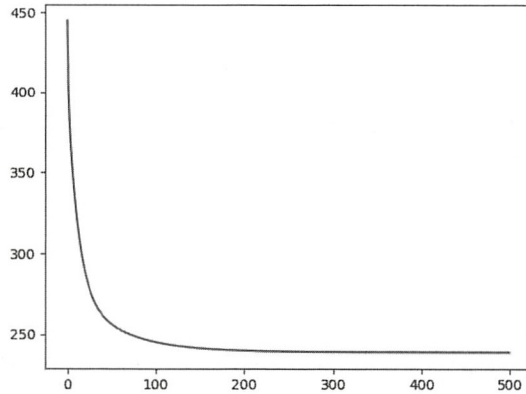


图 4-4 损失函数的值

通过以上程序，最终计算的超平面为 $z = 1.7042251x - 0.1592657y + 9.618724$ 。

2. 加载多元回归模型，并进行预测

我们已经将计算出的模型保存到了 ckpt 文件中，可以通过以下程序实现加载该文件中的模型，并计算该超平面在坐标 (6, 7) 和 (8, 10) 处的值，具体代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
import numpy as np
#"自变量(x,y)的值"
xy=tf.placeholder(tf.float32,[None,2])
#"初始化z=w1*x+w2*y+b中的w1、w2、b"
w=tf.Variable(tf.constant([[1],[1]],tf.float32),dtype=tf.float32,name='w')
b=tf.Variable(1.0,dtype=tf.float32,name='b')
#"超平面"
z=tf.matmul(xy,w)+b
#"声明一个tf.train.Saver类"
saver=tf.train.Saver()
#"从文件夹'./regression'下获得最近的ckpt文件"
```



```

ckpt=tf.train.latest_checkpoint('./regression')
#"打印返回的ckpt文件名"
print('获得的ckpt文件:'+ckpt)
#"创建会话"
session=tf.Session()
#"加载ckpt文件中的变量w和b的值"
saver.restore(session,ckpt)
#"计算在坐标(6,7)和(8,10)处的值"
pred=session.run(z,feed_dict={xy:np.array([[6,7],[8,10]],np.float32)})
print('在坐标(6,7)和(8,10)处的值:')
print(pred)

```

打印结果如下:

```

获得的ckpt文件:./regression/regressionModel.ckpt-499
在坐标(6,7)和(8,10)处的值:
[[18.729214]
 [21.659866]]

```

我们来总结利用 TensorFlow 处理回归问题的过程。我们将训练回归模型系统细分为以下 5 个步骤进行处理。

- (1) 输入已知数据 (即训练数据)。
- (2) 初始化需要的变量。
- (3) 根据已知数据和变量构造损失函数。
- (4) 选择某种梯度下降法。
- (5) 创建会话, 训练模型。

接下来, 我们用总结的这 5 个步骤处理非线性回归问题。

4.2 非线性回归分析

非线性回归与线性回归类似, 只是根据已知点拟合一条曲线或者曲面, 使得点到曲线或曲面的距离的平方和最小。仍以 4.1.3 节中提到的 xyz 三维空间中的 6 个点为例, 假设求以下形式的曲面:

$$z = (w_1x + w_2y)^2$$

使得这些点到曲面的距离的平方和最小，使用我们总结的 5 个步骤进行代码实现，具体如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
import numpy as np

#"第1步：输入已知数据"
x=tf.placeholder(tf.float32,[None])
y=tf.placeholder(tf.float32,[None])
z=tf.placeholder(tf.float32,[None])

#"第2步：初始化变量"
w1=tf.Variable(initial_value=2.0,dtype=tf.float32,name='w1')
w2=tf.Variable(initial_value=2.0,dtype=tf.float32,name='w2')

#"第3步：构造损失函数"
loss=tf.reduce_sum(tf.square(z-tf.pow((w1*x+w2*y),2.0)))

#"第4步：选用梯度下降法求解变量"
opti=tf.train.GradientDescentOptimizer(0.005).minimize(loss)

#"训练数据"
x_train=np.array([1,2,3,1,4,5],np.float32)
y_train=np.array([1,1,2,2,5,8],np.float32)
z_train=np.array([8,12,10,14,28,10],np.float32)

#"第5步：创建会话，训练模型"
session=tf.Session()
for i in range(500):
    session.run(opti,feed_dict={x:x_train,y:y_train,z:z_train})
```

也可以用另一种形式改进以上代码，因为函数 z 可以用矩阵的形式表示为

$$z = \left(\begin{bmatrix} x & y \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} \right)^2$$

可以将要求的变量 w_1 和 w_2 看成一个向量，令

$$\boldsymbol{w} = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}$$

具体实现代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
import numpy as np
```

```

#"第1步：输入已知数据"
xy=tf.placeholder(tf.float32,[None,2])
z=tf.placeholder(tf.float32,[None,1])
#"第2步：初始化变量"
w=tf.Variable(tf.constant([[1],[1]],tf.float32),dtype=tf.float32,name='w')
#"第3步：构造损失函数"
loss=tf.reduce_sum(tf.square(z-tf.matmul(xy,w)))
#"第4步：选用梯度下降法求解变量"
opti=tf.train.GradientDescentOptimizer(0.005).minimize(loss)
#"训练数据"
xy_train=np.array([[1,1],[2,1],[3,2],
                    [1,2],[4,5],[5,8]
                    ],np.float32)
z_train=np.array([[8],[12],[10],[14],[28],[10]],np.float32)
#"第5步：创建会话，训练模型"
session=tf.Session()
for i in range(500):
    session.run(opti,feed_dict={xy:xy_train,z:z_train})

```

所以，同一个问题我们可以利用 TensorFlow 进行灵活的处理。本章主要介绍利用 TensorFlow 处理回归模型的具体步骤及其常用的函数。从第 5 章开始，我们介绍全连接神经网络及其 TensorFlow 的具体实现。

5

全连接神经网络

5.1 基本概念

人工神经网络（简称神经网络）这个词儿听起来像是用人工的办法模拟人脑，加上它使用了生物有关的名词（比如“神经元”等），让人感觉它很神秘，并且可能让人联想到仿生学或认知科学等一辈子都搞不懂的东西。其实，除了借用生物学上的名词，并且做了一些形象的比喻，人工神经网络和人脑没有半点关系^[1]。人工神经网络可以简单地理解为一种特殊的针对向量的变换，常用数符号 $f: \mathbf{R}^N \rightarrow \mathbf{R}^M$ 表示 N 维向量经过变换 f 转换为 M 维向量。我们以二维向量转换为三维向量为例，假设变换 f 为

$$(x_1, x_2) \Rightarrow (2x_1 + x_2 + 1, x_1 + 2x_2 + 2, x_1 + x_2)$$

则向量 $(1, 2)$ 经过变换，转换为向量 $(2 \times 1 + 2 + 1, 1 + 2 \times 2 + 2, 1 + 2) = (5, 7, 3)$ 。

全连接神经网络就是一种变换的规则，只是这种规则比较复杂，单纯利用数学公式表示这种复杂的变换不够形象也不容易理解，一般用图的形式表示该变换，可以概括为如下两点。

(1) 如图 5-1 所示，图中所有节点都是分层的，为了便于交流，一些书和论文中使用了约定俗成的提法，比如最左边一层为输入层，最右边一层为输出层，中间各层被称为中间层或者隐含层，这里只画出了一个隐含层，理论上隐含层的层数是任意的。图 5-1 中的输入层有 2 个节点，中间层有 3 个节点，输出层有 2 个节点，节点在神经网络中又被称为神经元，可以将输入层的 2 个节点想象成长度为 2 的向量；将中间层的 3 个节点理解为从长度为 2 的向量变换成长度为 3 的向量。从中间层到输出层，就是将长度为 3 的向量转换成长度为 2 的

向量。

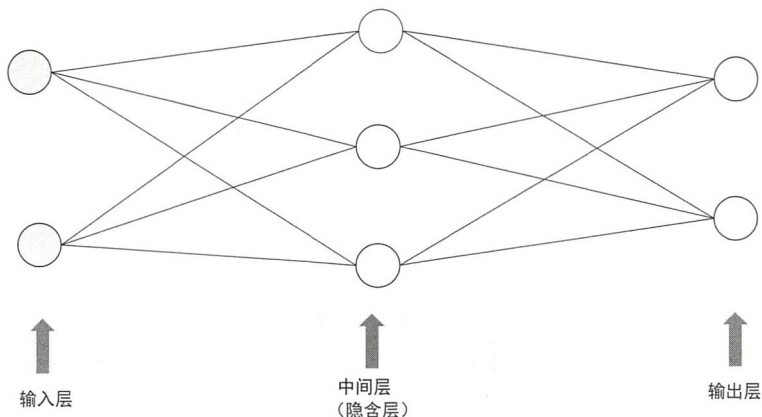


图 5-1 典型的三层全连接神经网络

(2) 从图 5-1 可以看出，每一层的神经元都和下一层的神经元通过一条弧进行连接，神经元与神经元之间连接的弧上都有一个值，常称为权重或者权值，常用符号 w 表示。所有指向同一个神经元的弧上，还有一个公有的值，常称为偏置，常用符号 b 表示。为了更方便地用数学符号表示权重和偏置，我们将输入层看作第 0 层，依此类推，后面就定义为第 1 层、第 2 层、第 n 层，这样就可以非常方便地定义符号 $w_{ij}^{(n)}$ 代表第 $n-1$ 层的第 i 个神经元到第 n 层的第 j 个神经元的权重， $b_i^{(n)}$ 代表第 n 层的第 i 个神经元的偏置，如图 5-2 所示。

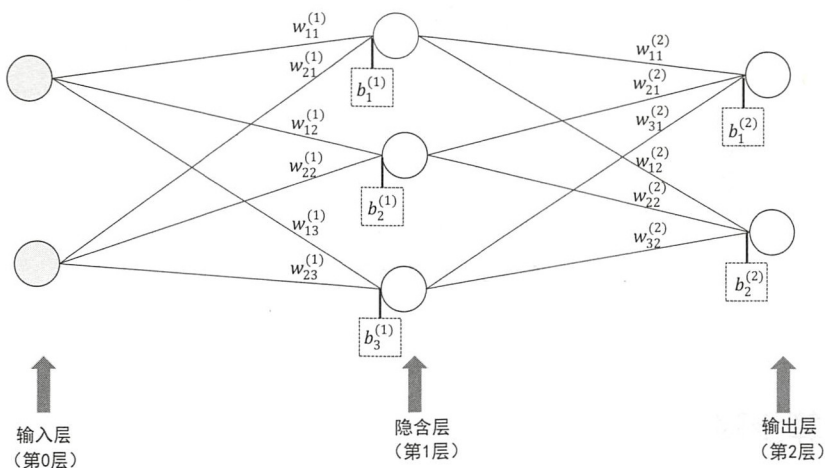


图 5-2 权重及其偏置

也可以利用矩阵的形式管理每一层的权重和偏置，如利用符号 $\mathbf{W}^{(n)}$ 代表第 n 层的权重矩阵，用来存储第 $n-1$ 层的所有神经元到第 n 层的所有神经元的权重， $\mathbf{b}^{(n)}$ 代表第 n 层的偏

置。图 5-2 中的全连接神经网络的第 1 层的权重矩阵及其偏置表示为

$$\mathbf{W}^{(1)} = \begin{bmatrix} w_{11}^{(1)} & w_{12}^{(1)} & w_{13}^{(1)} \\ w_{21}^{(1)} & w_{22}^{(1)} & w_{23}^{(1)} \end{bmatrix}, \quad \mathbf{b}^{(1)} = \begin{bmatrix} b_1^{(1)} \\ b_2^{(1)} \\ b_3^{(1)} \end{bmatrix}$$

显然，其中 $\mathbf{W}^{(1)}$ 的第 1 列代表第 0 层的所有神经元到第 1 层的第 1 个神经元的权重。同理，第 2 层的权重矩阵和偏置为

$$\mathbf{W}^{(2)} = \begin{bmatrix} w_{11}^{(2)} & w_{12}^{(2)} \\ w_{21}^{(2)} & w_{22}^{(2)} \\ w_{31}^{(2)} & w_{32}^{(2)} \end{bmatrix}, \quad \mathbf{b}^{(2)} = \begin{bmatrix} b_1^{(2)} \\ b_2^{(2)} \end{bmatrix}$$

根据权重和偏置及输入层的值，我们可以用非常简单的公式（甚至只是用一些简单的加法和乘法）顺序计算出每一层所有神经元处的值。至于如何计算整个神经网络中每个神经元处的值，我们用以下例子进行详细说明，对于三层全连接神经网络结构，假设权重和偏置的值如图 5-3 所示，接下来我们详细介绍关于该网络的计算步骤。

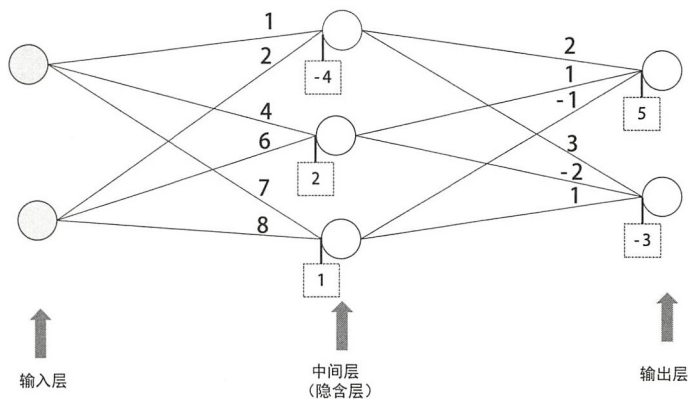


图 5-3 权重及其偏置

5.2 计算步骤

假设输入层神经元的值为 $\begin{bmatrix} 3 \\ 5 \end{bmatrix}$ ，首先计算隐含层（第 1 层）的第 1 个神经元的值，步骤如下。

第 1 步：第 0 层的所有神经元与该神经元进行连接，且已知它们之间的权重及其偏置，根据这些值计算线性组合：

$$l_1^{(1)} = 3 \times 1 + 5 \times 2 + (-4) = 9$$

第 2 步：将第 1 步得到的线性组合的值作为一个一元函数的输入，假设该函数为 $f(x) = 2x$ ，那么

$$\sigma_1^{(1)} = f(l_1^{(1)}) = 2 \times l_1^{(1)} = 2 \times 9 = 18$$

18 即当输入是 $\begin{bmatrix} 3 \\ 5 \end{bmatrix}$ 时，隐含层的第 1 个神经元处的值，如图 5-4 所示。

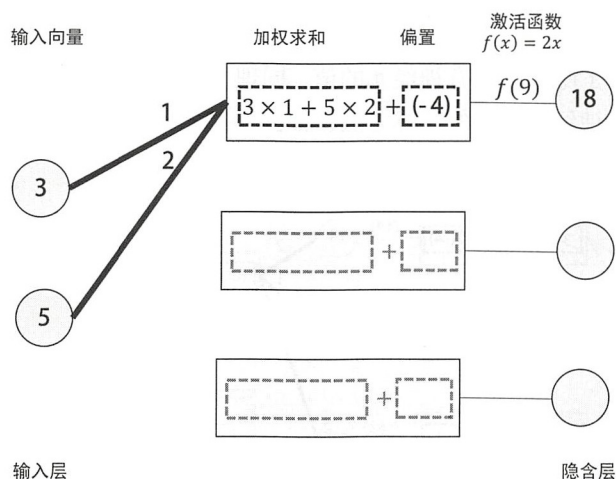


图 5-4 隐含层第 1 个神经元的值

上述函数在神经网络中被称为**激活函数**或**神经元函数**，本书后面章节会详细介绍常用的激活函数及其对应的性质。为了方便介绍，作者先解释后面章节常用的数学符号。符号 $l_i^{(n)}$ 代表第 n 层的第 i 个神经元的线性组合， $\sigma_i^{(n)}$ 代表对应的线性组合 $l_i^{(n)}$ 输入到激活函数后的返回值，即第 n 层第 i 个神经元的值，比如上述的 $l_1^{(1)}$ 代表第 1 层的第 1 个神经元的线性组合的值， $\sigma_1^{(1)}$ 代表 $l_1^{(1)}$ 输入到激活函数的返回值，即第 1 层第 1 个神经元的值。

计算隐含层的第 2 个神经元的方法同计算隐含层的第 1 个神经元的类似，如图 5-5 所示。计算隐含层的第 3 个神经元的方法，如图 5-6 所示。

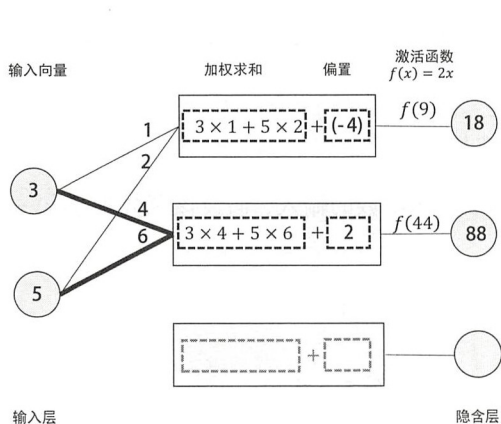


图 5-5 隐含层第 2 个神经元的值

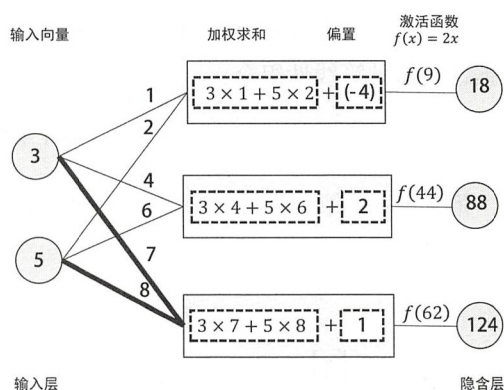


图 5-6 隐含层第 3 个神经元的值

以上已经计算出了隐含层所有神经元的值。同理，我们来计算输出层的每一个神经元的值。输出层的第 1 个神经元的值的计算方法，如图 5-7 所示。

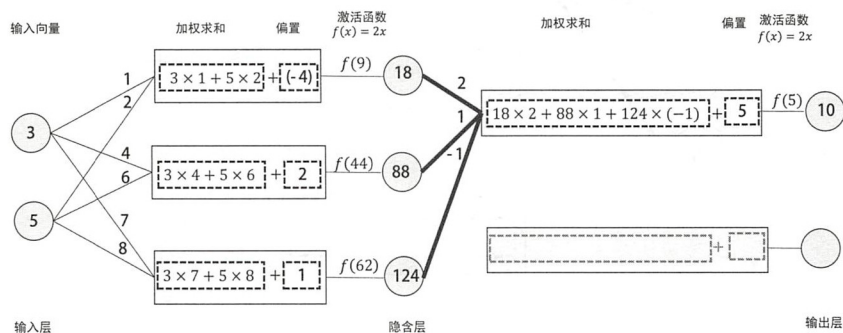


图 5-7 输出层第 1 个神经元的值

依此类推，输出层的第 2 个神经元的值的计算方法，如图 5-8 所示。

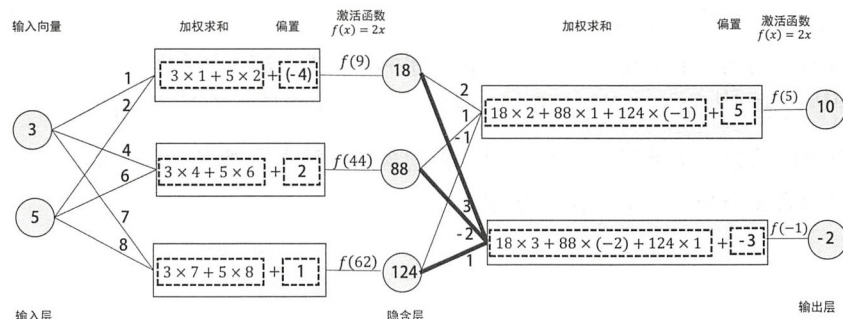


图 5-8 输出层第 2 个神经元的值

即当输入层的值是 $\begin{bmatrix} 3 \\ 5 \end{bmatrix}$ 时，通过该网络后变换为 $\begin{bmatrix} 10 \\ -2 \end{bmatrix}$ 。已知权重和偏置的全连接神经网络，因为它根据上一层的神经元的值计算下一层的神经元的值，所以又称为**前馈全连接神经网络**。

5.3 神经网络的矩阵表达

我们仍以 5.2 节中的三层全连接神经网络为例，介绍如何利用矩阵的形式，计算每一层的神经元的值，具体过程如下。

1. 从输入层到隐含层的矩阵表达

第 1 步：线性组合（加权求和及偏置）。

$$\mathbf{l}^{(1)} = (\mathbf{w}^{(1)})^T \mathbf{x} + \mathbf{b}^{(1)} = \begin{bmatrix} 1 & 2 \\ 4 & 6 \\ 7 & 8 \end{bmatrix} \begin{bmatrix} 3 \\ 5 \end{bmatrix} + \begin{bmatrix} -4 \\ 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 9 \\ 44 \\ 62 \end{bmatrix}$$

第 2 步：将第 1 步得到的线性组合作为激活函数的输入，即可得到第 1 层的每个神经元的值。

$$\boldsymbol{\sigma}^{(1)} = f \left(\begin{bmatrix} 9 \\ 44 \\ 62 \end{bmatrix} \right) = \begin{bmatrix} f(9) \\ f(44) \\ f(62) \end{bmatrix} = \begin{bmatrix} 18 \\ 88 \\ 124 \end{bmatrix}$$

2. 从隐含层到输出层的矩阵表达

第 1 步：线性组合（加权求和及偏置）。

$$\mathbf{l}^{(2)} = (\mathbf{w}^{(2)})^T \boldsymbol{\sigma}^{(1)} + \mathbf{b}^{(2)} = \begin{bmatrix} 2 & 1 & -1 \\ 3 & -2 & 1 \end{bmatrix} \begin{bmatrix} 18 \\ 88 \\ 124 \end{bmatrix} + \begin{bmatrix} 5 \\ -3 \end{bmatrix} = \begin{bmatrix} 5 \\ -1 \end{bmatrix}$$

第 2 步：将第 1 步得到的线性组合作为激活函数的输入，即可得到第 2 层的每个神经元的值。

$$\boldsymbol{\sigma}^{(2)} = f \left(\begin{bmatrix} 5 \\ -1 \end{bmatrix} \right) = \begin{bmatrix} f(5) \\ f(-1) \end{bmatrix} = \begin{bmatrix} 2 \times (5) \\ 2 \times (-1) \end{bmatrix} = \begin{bmatrix} 10 \\ -2 \end{bmatrix}$$

$\sigma^{(2)}$ 即当输入是 $\begin{bmatrix} 3 \\ 5 \end{bmatrix}$ 时，经过该网络的输出值。我们利用 TensorFlow 实现上述过程，代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
import numpy as np
#"输入层"
x=tf.placeholder(tf.float32,(2,None))
#"第1层的权重矩阵"
w1=tf.constant(
    [[1,4,7],
     [2,6,8]],tf.float32
)
#"第1层的偏置"
b1=tf.constant(
    [
     [-4],
     [2],
     [1]
    ],tf.float32
)
#"计算第1层的线性组合"
l1=tf.matmul(w1,x,True)+b1
#"激活2*x"
sigma1=2*l1
#"第2层的权重矩阵"
w2=tf.constant(
    [[2,3],
     [1,-2],
     [-1,1]
    ],tf.float32
)
#"第2层的偏置"
b2=tf.constant(
    [[5],[-3]],tf.float32
)
```

```

#"计算第1层的线性组合"
l2=tf.matmul(w2,sigma1,True)+b2
#"激活2*x"
sigma2=2*l2
#"创建会话"
session=tf.Session()
#"令x=[[3],[5]]"
print(session.run(sigma2,{x:np.array([[3],[5]],np.float32)}))

```

打印结果如下：

```

[[ 10.]
 [ -2.]]

```

以上示例只是介绍了一个输入（这里的输入指的是一个长度为2的向量）经过图5-3所示的全连接神经网络后得到输出值，在对应的实现代码中，我们注意到这个输入是按列存储的，利用矩阵相乘的函数 `matmul` 时，需要对权重矩阵转置，偏置需要存储在一个二维张量中。如果将神经网络的输入按行存储，则会更方便。以下计算多个输入分别经过该神经网络时对应的输出，用矩阵管理多个输入，其中每一个输入按行存储，假设有4个输入，如：

$$\mathbf{x} = \begin{bmatrix} 10 & 11 \\ 20 & 21 \\ 30 & 31 \\ 40 & 41 \end{bmatrix}$$

首先计算每一个输入分别在隐含层的线性组合，可以用如下矩阵的形式计算。

$$\mathbf{l}^{(1)} = \mathbf{x}\mathbf{w}^{(1)} + \mathbf{b}^{(1)} = \begin{bmatrix} 10 & 11 \\ 20 & 21 \\ 30 & 31 \\ 40 & 41 \end{bmatrix} \begin{bmatrix} 1 & 4 & 7 \\ 2 & 6 & 8 \end{bmatrix} + \begin{bmatrix} -4 & 2 & 1 \end{bmatrix} = \begin{bmatrix} 33 & 110 & 165 \\ 63 & 210 & 315 \\ 93 & 310 & 465 \\ 123 & 410 & 615 \end{bmatrix}$$

上述结果 $\mathbf{l}^{(1)}$ 的每一行分别代表每一个输入在隐含层的线性组合值。然后输入到激活函数，得到每一个输入在隐含层的值，即

$$\sigma^{(1)} = f(\mathbf{l}^{(1)}) = 2 \times \mathbf{l}^{(1)} = 2 \times \begin{bmatrix} 33 & 110 & 165 \\ 63 & 210 & 315 \\ 93 & 310 & 465 \\ 123 & 410 & 615 \end{bmatrix} = \begin{bmatrix} 66 & 220 & 330 \\ 126 & 420 & 630 \\ 186 & 620 & 930 \\ 246 & 820 & 1230 \end{bmatrix}$$

计算出每一个输入在隐含层处的值，接着计算输出层的线性组合，即

$$l^{(2)} = \sigma^{(1)} w^{(2)} + b^{(2)} = \begin{bmatrix} 66 & 220 & 330 \\ 126 & 420 & 630 \\ 186 & 620 & 930 \\ 246 & 820 & 1230 \end{bmatrix} \begin{bmatrix} 2 & 3 \\ 1 & -2 \\ -1 & 1 \end{bmatrix} + \begin{bmatrix} 5 & -3 \end{bmatrix} = \begin{bmatrix} 27 & 85 \\ 47 & 165 \\ 67 & 245 \\ 87 & 325 \end{bmatrix}$$

然后，将上述结果作为激活函数的输入，得到输出层的值，即

$$\sigma^{(2)} = f(l^{(2)}) = 2 \times l^{(2)} = 2 \times \begin{bmatrix} 27 & 85 \\ 47 & 165 \\ 67 & 245 \\ 87 & 325 \end{bmatrix} = \begin{bmatrix} 54 & 170 \\ 94 & 330 \\ 134 & 490 \\ 174 & 650 \end{bmatrix}$$

$\sigma^{(2)}$ 的每一行代表 4 个输入分别经过全连接神经网络的输出值，即

$$\begin{aligned} \begin{bmatrix} 10 & 11 \end{bmatrix} &\Rightarrow \begin{bmatrix} 54 & 170 \end{bmatrix} \\ \begin{bmatrix} 20 & 21 \end{bmatrix} &\Rightarrow \begin{bmatrix} 94 & 330 \end{bmatrix} \\ \begin{bmatrix} 30 & 31 \end{bmatrix} &\Rightarrow \begin{bmatrix} 134 & 490 \end{bmatrix} \\ \begin{bmatrix} 40 & 41 \end{bmatrix} &\Rightarrow \begin{bmatrix} 174 & 650 \end{bmatrix} \end{aligned}$$

通过矩阵运算，我们可以方便快速地计算出不同输入经过同一个全连接神经网络的输出值，上述过程对应的代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
import numpy as np
#"输入层:每一个输入按行存储"
x=tf.placeholder(tf.float32,(None,2))
#"第1层的权重矩阵"
w1=tf.constant(
    [
        [1,4,7],
        [2,6,8],
    ],tf.float32
)
#"第1层的偏置"
b1=tf.constant([1,4,7],tf.float32)
```



```

#b1=tf.constant([[1,4,7]],tf.float32)#"以上b1可以这样写"
#"计算第1层的线性组合"
l1=tf.matmul(x,w1)+b1
#"激活2*x"
sigma1=2*l1
#"第2层的权重矩阵"
w2=tf.constant(
    [[2,3],
     [1,-2],
     [-1,1]],tf.float32
)
#"第1层的偏置"
b2=tf.constant(
    [5,-3],tf.float32
)
#"计算第2层的线性组合"
l2=tf.matmul(sigma1,w2)+b2
#"激活2*x"
sigma2=2*l2
#"创建会话"
session=tf.Session()
#"4个输入"
print(session.run(sigma2,{x:np.array([
    [10,11],
    [20,21],
    [30,31],
    [40,41]],np.float32)})))

```

打印结果如下：

```

[[ 54. 170.]
 [ 94. 330.]
 [134. 490.]
 [174. 650.]]

```

上述代码中对全连接神经网络的输入是按行存储的，对比按列存储，这时偏置声明为一个一维张量即可，这样就充分利用了第2章介绍的 TensorFlow 实现的加法运算的优势。

通过上述示例，我们对全连接神经网络的计算形式有了比较完整的理解。5.4 节将介绍全连接神经网络中常用的激活函数，我们先大体介绍其函数的形式及其对应的 TensorFlow 实现，并从定义域、值域、导数这几个方面了解函数的性质，至于神经网络的优缺点，将在第 6 章介绍。

5.4 激活函数

激活函数是神经网络的重要组成部分，为了保证神经网络的灵活性及其计算的复杂度，激活函数一般不会太复杂，以下介绍常用的激活函数。

5.4.1 sigmoid 激活函数

sigmoid 激活函数的形式如下：

$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

其导数为

$$\text{sigmoid}'(x) = \frac{e^{-x}}{(1 + e^{-x})^2}$$

显然， $\text{sigmoid}(x) \in (0, 1)$ ， $\text{sigmoid}'(x) \in (0, 0.25)$ ，如图 5-9 所示。

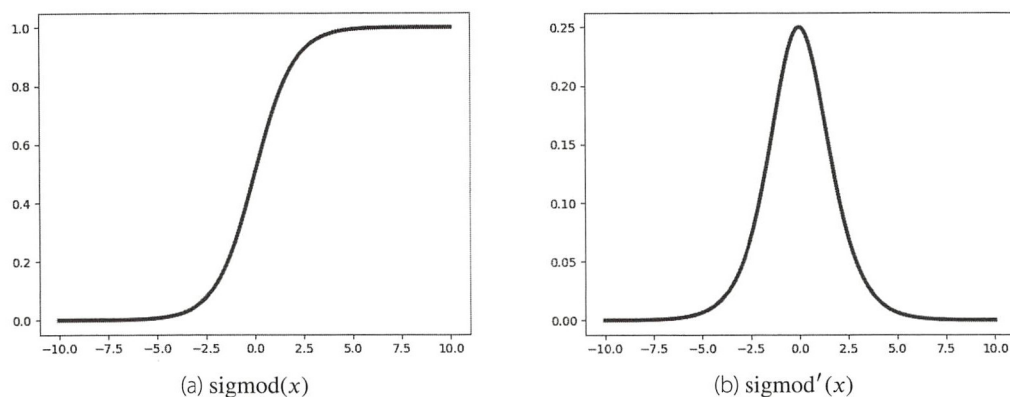


图 5-9 sigmoid 激活函数及其导数

TensorFlow 通过函数 `tf.nn.sigmoid(x, name=None)` 实现 sigmoid 激活函数，使用示例如下：

```

# -*- coding: utf-8 -*-
import tensorflow as tf
#"二维张量"
t=tf.constant([[1,3],[2,0]],tf.float32)
#"sigmod激活"
result=tf.nn.sigmoid(t)
#"创建会话"
session=tf.Session()
#"打印结果"
print(session.run(result))

```

打印结果如下:

```

[[ 0.7310586   0.95257413]
 [ 0.88079703   0.5       ]]

```

5.4.2 tanh 激活函数

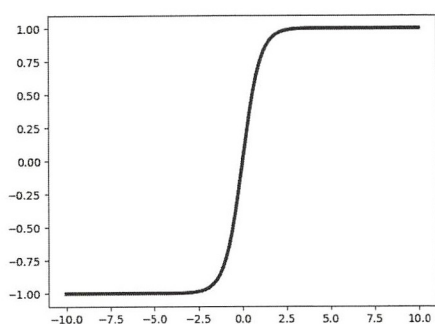
tanh 激活函数的形式如下:

$$\tanh(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}} = \frac{2}{1 + e^{-2x}} - 1$$

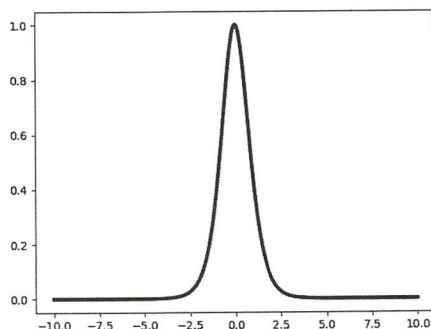
其导数为

$$\tanh'(x) = \frac{4e^{-2x}}{(1 + e^{-2x})^2} = 1 - (\tanh(x))^2$$

显然, $\tanh(x) \in (-1, 1)$, $\tanh'(x) \in (0, 1]$, 如图 5-10 所示。



(a) $\tanh(x)$



(b) $\tanh'(x)$

图 5-10 tanh 激活函数及其导数

TensorFlow 通过函数 `tf.nn.tanh(x, name=None)` 实现 `tanh` 激活函数，使用示例与函数 `tf.nn.sigmoid` 类似。

5.4.3 ReLU 激活函数

ReLU^[2] 激活函数的形式如下：

$$\text{relu}(x) = \max(x, 0) = \begin{cases} x, & x \geq 0 \\ 0, & x < 0 \end{cases}$$

其导数为

$$\text{relu}'(x) = \begin{cases} 1, & x \geq 0 \\ 0, & x < 0 \end{cases}$$

显然， $\text{relu}(x) \in [0, +\infty]$ ， $\text{relu}'(x) \in \{0, 1\}$ ，如图 5-11 所示。

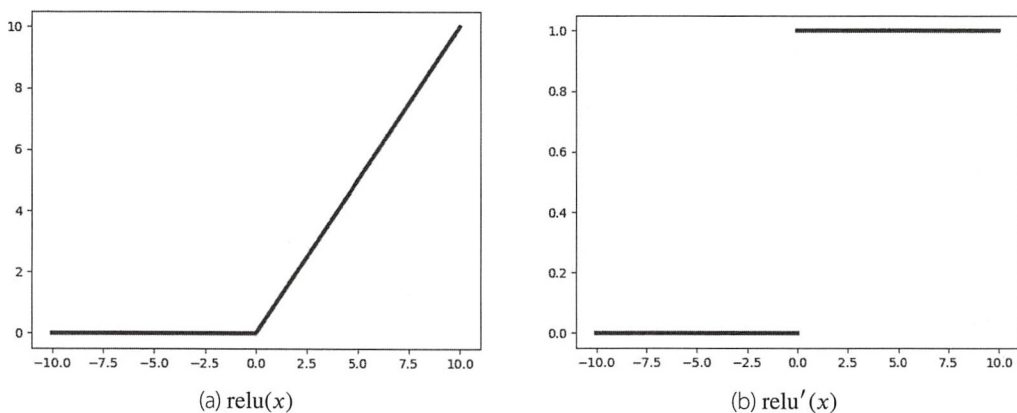


图 5-11 ReLU 激活函数及其导数

TensorFlow 通过函数 `tf.nn.relu(features, name=None)` 实现 ReLU 激活函数。

我们通过以下示例来了解 ReLU 函数的梯度。假设有三元函数

$$g(x_1, x_2, x_3) = 2x_1 + 3x_2 + 4x_3$$

函数 f 是关于 g 的复合函数，有

$$f(x_1, x_2, x_3) = \text{relu}(g) = \text{relu}(2x_1 + 3x_2 + 4x_3)$$

计算 f 在 $(x_1, x_2, x_3) = (2, 1, 3)$ 处的导数, $g(2, 1, 3) = 19$, $\frac{\partial f}{\partial g}|_{g=19} = 1$, 又根据导数的链式法则, 有

$$\frac{\partial f}{\partial x_1}|_{x_1=2} = \frac{\partial f}{\partial g}|_{g=19} \frac{\partial g}{\partial x_1}|_{x_1=2} = 1 \times 2 = 2$$

$$\frac{\partial f}{\partial x_2}|_{x_2=2} = \frac{\partial f}{\partial g}|_{g=19} \frac{\partial g}{\partial x_2}|_{x_2=1} = 1 \times 3 = 3$$

$$\frac{\partial f}{\partial x_3}|_{x_3=2} = \frac{\partial f}{\partial g}|_{g=19} \frac{\partial g}{\partial x_3}|_{x_3=3} = 1 \times 4 = 4$$

我们利用计算梯度的函数 `gradients` 实现以上示例, 代码如下:

```
# -*- coding: utf-8 -*-
import tensorflow as tf
#"变量"
x=tf.Variable(tf.constant([[2,1,3]],tf.float32))
w=tf.constant([[2],[3],[4]],tf.float32)
#"函数g"
g=tf.matmul(x,w)
#"函数f=relu(g)"
f=tf.nn.relu(g)
#"计算f在(2,1,3)处的导数"
gradient=tf.gradients(f,[g,x])
session=tf.Session()
session.run(tf.global_variables_initializer())
#"打印结果"
print(session.run(gradient))
```

打印结果如下:

```
[array([[1.]], dtype=float32), array([[2., 3., 4.]], dtype=float32)]
```

5.4.4 leaky relu 激活函数

leaky relu^[3] 激活函数的形式如下:

$$\text{leakyrelu}(x) = \begin{cases} x, & x \geq 0 \\ \alpha x, & x < 0 \end{cases}$$

其导数为

$$\text{leakyrelu}'(x) = \begin{cases} 1, & x \geq 0 \\ \alpha, & x < 0 \end{cases}$$

一般令 $\alpha = 0.01$ 。TensorFlow 通过函数 `tf.nn.leaky_relu(features, alpha=0.2, name=None)` 实现 leaky relu 函数，其中默认 $\alpha = 0.2$ 。显然， $\text{leakyrelu}(x) \in (-\infty, +\infty)$ ， $\text{leakyrelu}'(x) \in [1, \alpha]$ ，如图 5-12 所示。

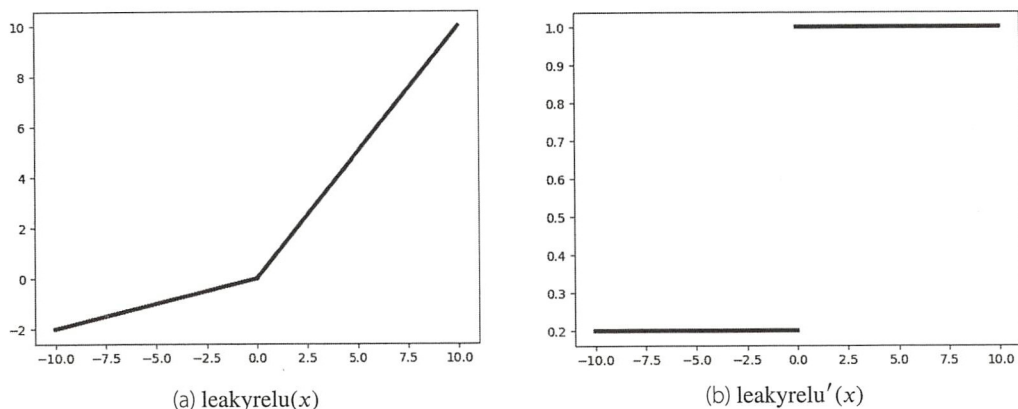


图 5-12 leaky relu 激活函数及其导数

我们通过一个梯度下降的简单示例来了解关于 leaky relu 函数的梯度。假设有一个三元函数

$$g(x_1, x_2, x_3) = 2x_1 + 3x_2 + 4x_3$$

函数 f 是关于 g 的符合函数，有

$$f(x_1, x_2, x_3) = \text{leakyrelu}(g) = \text{leakyrelu}(2x_1 + 3x_2 + 4x_3)$$

初始化 $(x_1, x_2, x_3) = (2, 1, 3)$ ，利用牛顿梯度下降法，计算 3 次迭代后 (x_1, x_2, x_3) 的值，其中令学习率 $\eta = 0.5$ 。

第 1 次迭代:

$$\frac{\partial f}{\partial g}|_{g=19} = 1, \quad \frac{\partial f}{\partial \mathbf{x}}|_{\mathbf{x}=(2,3,4)} = \begin{bmatrix} \frac{\partial f}{\partial x_1}|_{x_1=2} \\ \frac{\partial f}{\partial x_2}|_{x_2=1} \\ \frac{\partial f}{\partial x_3}|_{x_3=3} \end{bmatrix} = \begin{bmatrix} 2 \\ 3 \\ 4 \end{bmatrix}$$

$$\mathbf{x} \leftarrow \mathbf{x} - 0.5 \times \frac{\partial f}{\partial \mathbf{x}}|_{\mathbf{x}=(2,3,4)} = \begin{bmatrix} 2 \\ 3 \\ 4 \end{bmatrix} - 0.5 \times \begin{bmatrix} 2 \\ 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 1 \\ -0.5 \\ 1 \end{bmatrix}$$

第 2 次迭代:

$$g(1, -0.5, 1) = 2 \times 3 + 3 \times (-0.5) + 4 \times 1 = 2 - 1.5 + 4 = 4.5$$

$$\frac{\partial f}{\partial g}|_{g=4.5} = 1, \quad \frac{\partial f}{\partial \mathbf{x}}|_{\mathbf{x}=(1,-0.5,1)} = \begin{bmatrix} \frac{\partial f}{\partial x_1}|_{x_1=1} \\ \frac{\partial f}{\partial x_2}|_{x_2=-0.5} \\ \frac{\partial f}{\partial x_3}|_{x_3=1} \end{bmatrix} = \begin{bmatrix} 2 \\ 3 \\ 4 \end{bmatrix}$$

$$\mathbf{x} \leftarrow \mathbf{x} - 0.5 \times \frac{\partial f}{\partial \mathbf{x}}|_{\mathbf{x}=(1,-0.5,1)} = \begin{bmatrix} 1 \\ -0.5 \\ 1 \end{bmatrix} - 0.5 \times \begin{bmatrix} 2 \\ 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 0 \\ -2 \\ -1 \end{bmatrix}$$

第 3 次迭代:

$$g(0, -2, -1) = 2 \times 0 + 3 \times (-2) + 4 \times (-1) = -10$$

$$\frac{\partial f}{\partial g}|_{g=-10} = 0.2, \quad \frac{\partial f}{\partial \mathbf{x}}|_{\mathbf{x}=(0,-2,-1)} = \begin{bmatrix} \frac{\partial f}{\partial x_1}|_{x_1=0} \\ \frac{\partial f}{\partial x_2}|_{x_2=-2} \\ \frac{\partial f}{\partial x_3}|_{x_3=-1} \end{bmatrix} = \begin{bmatrix} 0.4 \\ 0.6 \\ 0.8 \end{bmatrix}$$

$$\mathbf{x} \leftarrow \mathbf{x} - 0.5 \times \frac{\partial f}{\partial \mathbf{x}}|_{\mathbf{x}=(0,-2,-1)} = \begin{bmatrix} 0 \\ -2 \\ -1 \end{bmatrix} - 0.5 \times \begin{bmatrix} 0.4 \\ 0.6 \\ 0.8 \end{bmatrix} = \begin{bmatrix} -0.2 \\ -2.3 \\ -1.4 \end{bmatrix}.$$

上述示例过程的代码如下:

图解深度学习与神经网络：从张量到 TensorFlow 实现

```

# -*- coding: utf-8 -*-
import tensorflow as tf
#"变量"
x=tf.Variable(tf.constant([[2,1,3]],tf.float32))
w=tf.constant([[2],[3],[4]],tf.float32)
#"函数g"
g=tf.matmul(x,w)
#"函数f=leaky_relu(g)"
f=tf.nn.leaky_relu(g,alpha=0.2)
#"牛顿梯度下降法"
opti=tf.train.GradientDescentOptimizer(0.5).minimize(f)
session=tf.Session()
session.run(tf.global_variables_initializer())
#"打印结果"
for i in range(3):
    session.run(opti)
    print('第%d次迭代的值'%(i+1))
    print(session.run(x))

```

打印结果如下：

```

"第1次迭代的值"
[[ 1. -0.5  1. ]]
"第2次迭代的值"
[[ 0. -2. -1.]]
"第3次迭代的值"
[[-0.2 -2.3 -1.4]]

```

5.4.5 elu 激活函数

elu^[4] 激活函数的形似如下：

$$\text{elu}(x) = \begin{cases} \alpha(e^x - 1), & x < 0 \\ x, & x \geq 0 \end{cases}$$

其导数为

$$\text{elu}'(x) = \begin{cases} \alpha e^x = \text{elu}(x) + \alpha, & x < 0 \\ 1, & x \geq 0 \end{cases}$$

TensorFlow 通过函数 `tf.nn.elu(features, name=None)` 实现当 $\alpha = 1$ 时的 `elu` 激活函数。显然, 当 $\alpha = 1$ 时, $\text{elu}(x) \in (-1, +\infty)$, $\text{elu}'(x) \in (0, 1]$, 如图 5-13 所示。

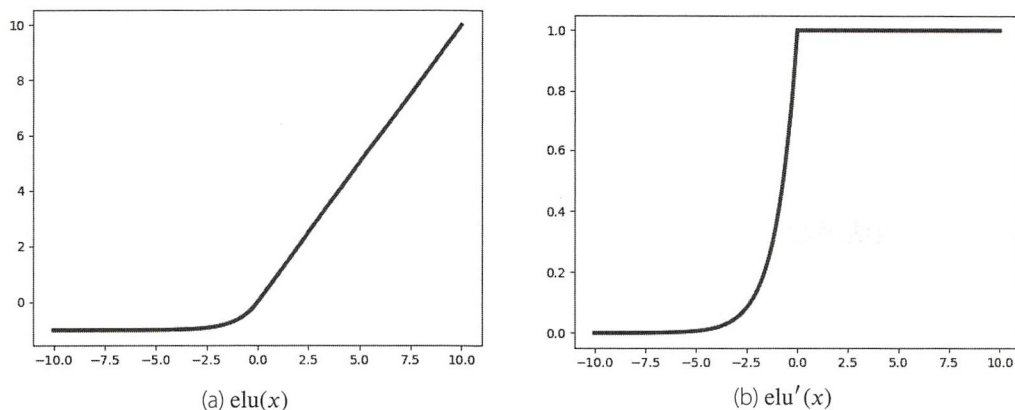


图 5-13 `elu` 激活函数及其导数

5.4.6 `crelu` 激活函数

`crelu`^[5] 激活函数的形式如下:

$$\text{crelu}(x) = (\max(x, 0), \max(-x, 0))$$

举一个关于 `crelu` 激活函数的简单示例, 如果输入值是 -2 , 则结果如下:

$$\text{crelu}(-2) = (\max(-2, 0), \max(2, 0)) = (0, 2)$$

再举个稍微复杂的例子, 如果输入值是一个一维张量, 则结果如下:

$$\text{crelu}([2, -1]) = (\max(2, 0), \max(-2, 0), \max(-1, 0), \max(1, 0)) = (2, 0, 0, 1)$$

TensorFlow 通过函数 `tf.nn.crelu(features, name=None)` 实现 `crelu` 激活函数, 该函数使用的示例代码如下:

```
# -*- coding: utf-8 -*-
import tensorflow as tf
#"输入张量"
t=tf.constant([-2,0,1],tf.float32)
```

图解深度学习与神经网络：从张量到 TensorFlow 实现

```
#"crelu激活函数"
r=tf.nn.crelu(t)
#"创建会话"
session=tf.Session()
#"打印结果"
print(session.run(r))
```

打印结果如下：

```
[ 0.  0.  1.  2. -0. -0.]
```

5.4.7 selu 激活函数

selu^[6] 激活函数的形式如下：

$$\text{selu}(x) = \begin{cases} \lambda \alpha (e^x - 1), & x < 0 \\ \lambda x, & x \geq 0 \end{cases}$$

其导数为

$$\text{selu}'(x) = \begin{cases} \lambda \alpha e^x, & x < 0 \\ \lambda, & x \geq 0 \end{cases}$$

一般 $\lambda = 1.0507$, $\alpha = 1.67326$, 如图 5-14 所示。

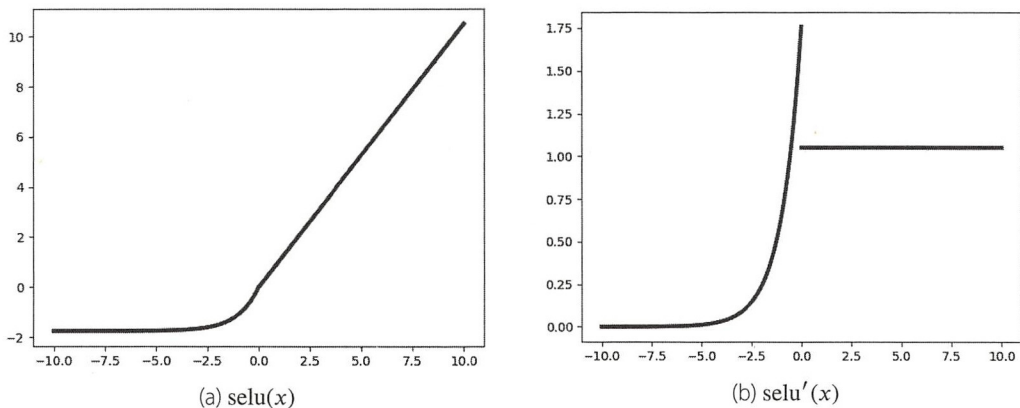


图 5-14 selu 激活函数及其导数

TensorFlow 通过函数 `tf.nn.selu(features, name=None)` 实现 selu 激活函数。

5.4.8 relu6 激活函数

relu6^[7] 激活函数的形式如下：

$$\text{relu6}(x) = \min(\max(x, 0), 6)$$

其导数为

$$\text{relu6}'(x) = \begin{cases} 1, & 0 < x < 6 \\ 0, & \text{其他} \end{cases}$$

显然, $\text{relu6}(x) \in [0, 6]$, $\text{relu6}(x) \in \{0, 1\}$, 如图 5-15 所示。

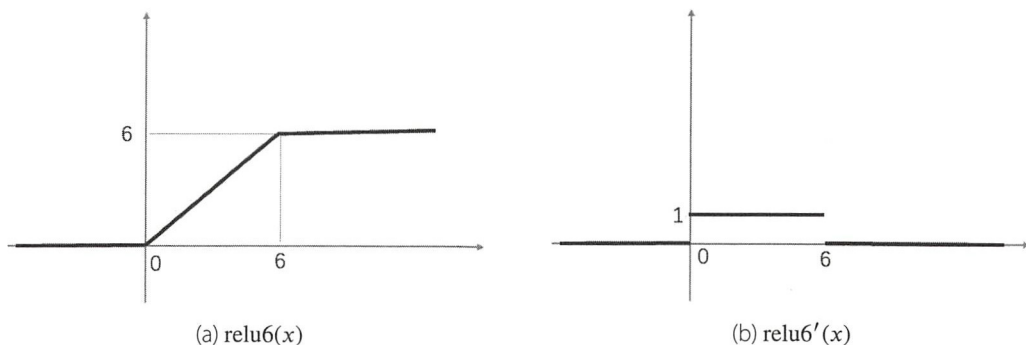


图 5-15 relu6 激活函数及其导数

TensorFlow 通过函数 `tf.nn.relu6(features, name=None)` 实现 relu6 激活函数。

5.4.9 softplus 激活函数

softplus^[8] 激活函数的形式如下：

$$\text{softplus}(x) = \ln(1 + \exp(x))$$

其导数为

$$\text{softplus}'(x) = \frac{\exp(x)}{1 + \exp(x)}$$

其中 $\text{softplus}(x) \in (0, +\infty)$, $\text{softplus}'(x) \in (0, 1)$, 如图 5-16 所示。

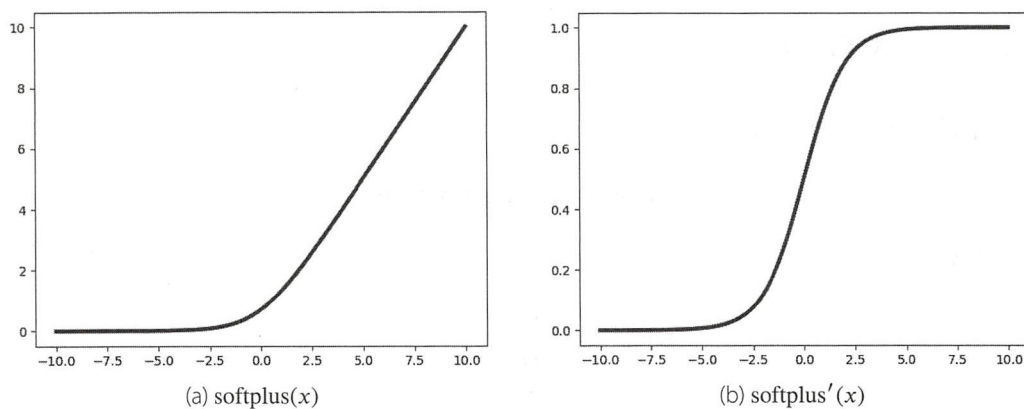


图 5-16 softplus 激活函数及其导数

TensorFlow 通过函数 `tf.nn.softplus(features, name=None)` 实现 `softplus` 激活函数。

5.4.10 softsign 激活函数

`softsign`^[9-10] 激活函数的形式如下：

$$\text{softsign}(x) = \frac{x}{1 + |x|}$$

其导数为

$$\text{softsign}'(x) = \frac{|x|}{1 + |x|^2}$$

显然， $\text{softsign}(x) \in (-1, +1)$ ， $\text{softsign}'(x) \in (0, +1)$ ，如图 5-17 所示。

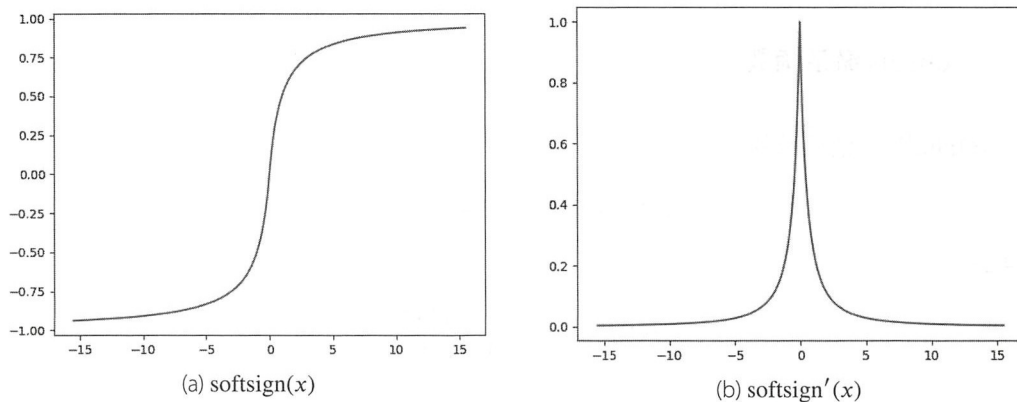


图 5-17 softsign 激活函数及其导数

TensorFlow 通过函数 `r=tf.nn.softsign(features, name=None)` 实现 `softsign` 激活函数。

对于神经网络的激活函数，现在使用比较多的是 `ReLU` 激活函数，除了本身及其导数的计算复杂度低，使用该函数作为神经网络常用的激活函数的优缺点，后续章节会根据具体问题具体分析。本章我们只需要掌握 `ReLU` 激活函数的形式和对应的 TensorFlow 函数接口即可。

至此，我们已对全连接神经网络的结构及其对应的 TensorFlow 实现有了全面的认识。第 6 章，我们将介绍如何利用神经网络处理机器学习中常遇到的分类问题，以及其对应的 TensorFlow 实现。

5.5 参考文献

- [1] 吴军.《数学之美》. 第二版. 北京: 人民邮电出版社, 2014.
- [2] Nair, Vinod; Hinton, Geoffrey E. (2010), “Rectified Linear Units Improve Restricted Boltzmann Machines”, 27th International Conference on International Conference on Machine Learning, ICML’10, USA: Omnipress, pp. 807-814, ISBN 9781605589077
- [3] Maas, Andrew L.; Hannun, Awni Y.; Ng, Andrew Y. (June 2013). “Rectifier nonlinearities improve neural network acoustic models” (PDF). Proc. ICML. 30 (1). Retrieved 2 January 2017
- [4] Clevert, Djork-Arné; Unterthiner, Thomas; Hochreiter, Sepp (2015-11-23). “Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)”. arXiv:1511.07289? Freely accessible [cs.LG].
- [5] Wenling Shang, Kihyuk Sohn, Diogo Almeida, Honglak Lee. Understanding and Improving Convolutional Neural Networks via Concatenated Rectified Linear Units. 2016
- [6] Klambauer, Günter; Unterthiner, Thomas; Mayr, Andreas; Hochreiter, Sepp (2017-06-08). “Self-Normalizing Neural Networks”.
- [7] Alex Krizhevsky. Convolutional Deep Belief Networks on CIFAR-10.
- [8] Glorot, Xavier; Bordes, Antoine; Bengio, Yoshua (2011). “Deep sparse rectifier neural networks” (PDF). International Conference on Artificial Intelligence and Statistics
- [9] Bergstra, James; Desjardins, Guillaume; Lamblin, Pascal; Bengio, Yoshua (2009). “Quadratic polynomials learn better image features”. Technical Report 1337”. Département d’Informatique et de Recherche Opérationnelle, Université de Montréal.

- [10] Glorot, Xavier; Bengio, Yoshua (2010), “Understanding the difficulty of training deep feedforward neural networks” (PDF), International Conference on Artificial Intelligence and Statistics (AISTATS’10), Society for Artificial Intelligence and Statistics

6

神经网络处理分类问题

本章将从管理数据、用数据建立模型、求解模型这几个方面完整地介绍如何利用 TensorFlow 处理分类问题。我们首先介绍利用 TensorFlow 管理数据的方法。

6.1 TFRecord 文件

TFRecord 是 TensorFlow 设计的一种存储数据的内置文件格式，可以方便高效地管理数据及这些数据的相关信息，利用它可以将数据快速加载到内存中。本节介绍如何将数据写入 TFRecord 文件和从 TFRecord 中解析数据的过程。

6.1.1 将 ndarray 写入 TFRecord 文件

我们通过以下示例理解如何将数据写入 TFRecord 文件。假设有 3 个三维的 ndarray，尺寸依次为 2 行 3 列 4 深度、3 行 3 列 3 深度和 2 行 2 列 3 深度，如图 6-1 所示。

图解深度学习与神经网络：从张量到 TensorFlow 实现

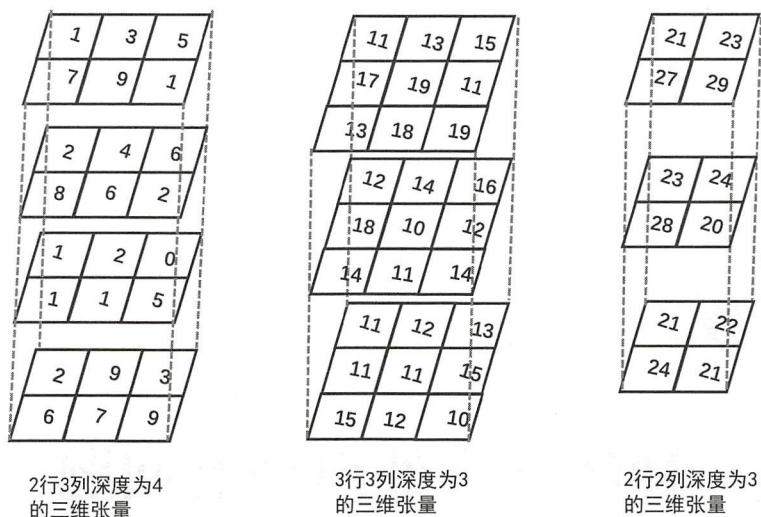


图 6-1 3 个三维 ndarray

将这 3 个 ndarray 及其对应的尺寸（高、宽、深度）写入文件名为 data.tfrecord 的 TFRecord 文件中，具体代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
import numpy as np
#"创建文件"
record=tf.python_io.TFRecordWriter('data.tfrecord')
#"高为2、宽为3、深度为4的三维ndarray"
array1=np.array(
    [
        [[1,2,1,2],[3,4,2,9],[5,6,0,3]],
        [[7,8,1,6],[9,6,1,7],[1,2,5,9]]
    ],np.float32
)
#"高为3、宽为3、深度为3的三维ndarray"
array2=np.array(
    [
        [[11,12,11],[13,14,12],[15,16,13]],
        [[17,18,11],[19,10,11],[11,12,15]],
        [[13,14,15],[18,11,12],[19,14,10]]
    ],np.float32
)
```

```

    )

#"高为2、宽为2、深度为3的三维ndarray"
array3=np.array(
    [
        [[21,23,21],[23,24,22]],
        [[27,28,24],[29,20,21]]
    ],np.float32
)

#"将上述3个ndarray存入一个列表"
arrays=[array1,array2,array3]
#"循环处理上述列表中的每一个ndarray"
for array in arrays:
    #"计算每一个ndarray的形状（高、宽、深度）"
    height,width,depth=array.shape
    #"将ndarray中的值转为字节类型"
    array_raw=array.tostring()
    #"ndarray的值及对应的高、宽、深度"
    feature={
        'array_raw':
            tf.train.Feature(
                bytes_list=tf.train.BytesList(value=[array_raw])),
        'height':
            tf.train.Feature(int64_list=tf.train.Int64List(value=[height])),
        'width':
            tf.train.Feature(int64_list=tf.train.Int64List(value=[width])),
        'depth':
            tf.train.Feature(int64_list=tf.train.Int64List(value=[depth]))
    }
    features=tf.train.Features(feature=feature)
    example=tf.train.Example(features=features)
    #"字符串序列化后写入文件"
    record.write(example.SerializeToString())
record.close()

```

运行上述程序，在当前目录下生成一个名称为 data.tfrecord 的 TFRecord 文件。在上述程序中，首先得到每一个 ndarray 的高、宽和深度，然后利用 ndarray 的成员函数 `tostring()` 将 ndarray 转换为字节类型，并将这些数据存储在 `tf.train.Example` 对象中，再利用其成员

函数 `SerializeToString()` 序列化为二进制字符串，最后利用 `TFRecordWriter` 对象的成员函数 `write` 将序列化后的二进制字符串写入 `TFRecord` 文件中。熟悉了如何写入 `TFRecord` 文件后，我们将介绍如何从 `TFRecord` 文件中解析（读取）数据。

6.1.2 从 TFRecord 解析数据

我们已经了解了如何将数据保存到 `TFRecord` 文件中，接下来我们介绍如何从 `TFRecord` 文件中解析（读取）数据。该过程与写入文件的过程相反，首先利用函数 `tf.train.string_input_producer` 读取一个 `TFRecord` 文件列表，然后创建一个 `TFRecordReader` 对象，利用其成员函数读取 `TFRecord` 文件列表，最后利用函数 `tf.parse_single_example` 解析文件中的数据。我们可以根据不同的需求从 `TFRecord` 文件中顺序抽取数据，也可以随机抽取数据，该过程的详细介绍如下。

1. 从 TFRecord 文件中顺序解析数据

接着 6.1.1 节的示例，从生成的 `data.tfrecord` 文件中顺序读取数据，具体代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
import numpy as np
#"读取tfrecord文件列表，这里只有一个tfrecord"
records_queue=tf.train.string_input_producer(['data.tfrecord'],num_epochs=2)
#"创建一个TFRecordReader对象"
reader=tf.TFRecordReader()
_,serialized_example=reader.read(records_queue)
#"解析tfrecord中的数据，每次只解析一个"
features=tf.parse_single_example(
    serialized_example,
    features={
        'array_raw':tf.FixedLenFeature([],tf.string),
        'height':tf.FixedLenFeature([],tf.int64),
        'width':tf.FixedLenFeature([],tf.int64),
        'depth':tf.FixedLenFeature([],tf.int64)
    }
)
#"解析出对应的值"
```



```

array_raw=features['array_raw']
array = tf.decode_raw(array_raw,tf.float32)#"解码"
height= features['height']
width = features['width']
depth = features['depth']
#"创建会话"
session=tf.Session()
session.run(tf.global_variables_initializer())
session.run(tf.local_variables_initializer())
coord=tf.train.Coordinator()
threads=tf.train.start_queue_runners(sess=session,coord=coord)
#"循环5次解析文件流中的数据"
for i in range(5):
    ndarray,h,w,d=session.run([array,height,width,depth])
    print('---第%(num)d次解析到的ndarray---'%(num':i+1})
    print(ndarray)
coord.request_stop()
coord.join(threads)
session.close()

```

上述代码中，循环 5 次从 TFRecord 文件中解析的数据。当然，每次只能解析 1 个，打印结果如下。

```

---"第1次解析到的ndarray"---
[1. 2. 1. 2. 3. 4. 2. 9. 5. 6. 0. 3. 7. 8. 1. 6. 9. 6. 1. 7. 1. 2. 5. 9.]
---"第2次解析到的ndarray"---
[11. 12. 11. 13. 14. 12. 15. 16. 13. 17. 18. 11. 19. 10. 11. 11. 12. 15.
 13. 14. 15. 18. 11. 12. 19. 14. 10.]
---"第3次解析到的ndarray"---
[21. 23. 21. 23. 24. 22. 27. 28. 24. 29. 20. 21.]
---"第4次解析到的ndarray"---
[1. 2. 1. 2. 3. 4. 2. 9. 5. 6. 0. 3. 7. 8. 1. 6. 9. 6. 1. 7. 1. 2. 5. 9.]
---"第5次解析到的ndarray"---
[11. 12. 11. 13. 14. 12. 15. 16. 13. 17. 18. 11. 19. 10. 11. 11. 12. 15.
 13. 14. 15. 18. 11. 12. 19. 14. 10.]

```

从打印结果可以看出，解析出的每个 ndarray 都是一维的，这是因为在写入文件时，首先将原数据转换为字节类型。当然，我们也可以根据解析到的每个 ndarray 对应的高、宽和

深度将其转换为三维的 ndarray，代码如下：

#"循环5次解析文件流中的数据"

```
for i in range(5):
    ndarray,h,w,d=session.run([array,height,width,depth])
    ndarray=np.reshape(ndarray,[h,w,d])
    print('---第%(num)d次解析到的ndarray---'%{'num':i+1})
    print(ndarray)
```

打印结果如下：

```
--- 第1次解析到的ndarray ---
[[[1. 2. 1. 2.],[3. 4. 2. 9.],[5. 6. 0. 3.]],
 [[7. 8. 1. 6.],[9. 6. 1. 7.],[1. 2. 5. 9.]]]
--- 第2次解析到的ndarray ---
[[[11. 12. 11.],[13. 14. 12.],[15. 16. 13.]],
 [[17. 18. 11.],[19. 10. 11.],[11. 12. 15.]],
 [[13. 14. 15.],[18. 11. 12.],[19. 14. 10.]]]
--- 第3次解析到的ndarray ---
[[[21. 23. 21.],[23. 24. 22.]],
 [[27. 28. 24.],[29. 20. 21.]]]
--- 第4次解析到的ndarray ---
[[[1. 2. 1. 2.],[3. 4. 2. 9.],[5. 6. 0. 3.]],
 [[7. 8. 1. 6.],[9. 6. 1. 7.],[1. 2. 5. 9.]]]
--- 第5次解析到的ndarray---
[[[11. 12. 11.],[13. 14. 12.],[15. 16. 13.]],
 [[17. 18. 11.],[19. 10. 11.],[11. 12. 15.]]
 [[13. 14. 15.],[18. 11. 12.],[19. 14. 10.]]]
```

注意：我们只在 TFRecord 文件中写入了 3 个 ndarray，为什么可以循环解析 5 次，甚至更多次呢？这是由于函数 `tf.train.string_input_producer` 中的参数 `num_epochs` 设置的原因，可以简单地理解为解析完所有的数据代表 1 epoch（轮），这里设置为 2，意思是可以对整个数据解析 2 轮。如果设置为 1，就会报出越界的错误。因为设置为 1，代表只能解析 1 轮，1 轮就是 3 个 ndarray，所以不能解析 5 次。

以上示例是从 TFRecord 文件中顺序读取数据，接下来，作者介绍如何从 TFRecord 文件中随机解析数据。

2. 从 TFRecord 文件中随机解析数据

在 6.1.1 节的示例中，写入 TFRecord 文件的 3 个 ndarray 的尺寸是不同的，大部分情况下，我们用 TFRecord 文件管理数据，管理的都是相同尺寸的数据，这样更方便以多种形式解析数据。如图 6-2 所示，有 3 个尺寸同为 1 行 2 列 3 深度的 ndarray，写入文件名为 dataTest.tfrecord 的 TFRecord 文件中，因为已经知道这 3 个 ndarray 的尺寸都是 1 行 2 列 2 深度，所以一般只存储原数据即可，不存储其高、宽、深度等信息，代码同 6.1.1 节的类似，具体代码如下：

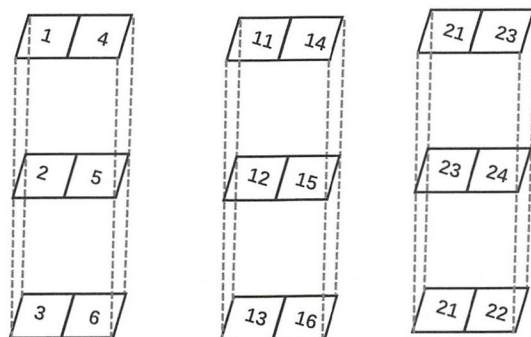


图 6-2 3 个相同尺寸的三维 ndarray

```
# -*- coding: utf-8 -*-
import tensorflow as tf
import numpy as np
#"创建文件"
record=tf.python_io.TFRecordWriter('dataTest.tfrecord')
#"高为1、宽为2、深度为3的三维ndarray"
array1=np.array(
    [
        [[1,2,3],[4,5,6]],
    ],np.float32
)
#"高为1、宽为2、深度为3的三维ndarray"
array2=np.array(
    [
        [[11,12,13],[14,15,16]],
    ],np.float32
)
```

```
#"高为1、宽为2、深度为3的三维ndarray"
array3=np.array(
    [
        [[21,23,21],[23,24,22]],
    ],np.float32
)

#"将上述3个ndarray存入一个列表"
arrays=[array1,array2,array3]
#"循环处理上述列表中的每一个ndarray"
for array in arrays:
    #"将ndarray中的值转为字节类型"
    array_raw=array.tostring()
    #"ndarray的值"
    feature={
        'array_raw':
            tf.train.Feature(bytes_list=tf.train.
                ByteList(value=[array_raw])),
    }
    features=tf.train.Features(feature=feature)
    example=tf.train.Example(features=features)
    #"字符串序列化后写入文件"
    record.write(example.SerializeToString())
record.close()
```

在介绍从 TFRecord 文件顺序解析数据时，我们只能每次从 TFRecord 文件中解析一个数据。接下来，我们每次从 TFRecord 文件中读取多个数据，并且是随机读取，不是按顺序读取的，以每次读取 2 个数据为例，具体代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
#"创建TFRecordReader对象"
epochs=2
reader=tf.TFRecordReader()
records_queue=tf.train.string_input_producer(['dataTest.tfrecord'],
                                              num_epochs=epochs)
_,serialized_example=reader.read(records_queue)
#"解析文件中的图像及其对应的标签"
```

```

features=tf.parse_single_example(
    serialized_example,
    features={
        'array_raw':tf.FixedLenFeature([],tf.string)
    }
)

#"解码二进制数据"
array_raw=features['array_raw']
array_raw=tf.decode_raw(array_raw,tf.float32)
array=tf.reshape(array_raw,[1,2,3])
#"每次从文件中读取2个数据"
BatchSize =2#"不能大于文件中数据的个数"
arrays=tf.train.shuffle_batch([array],BatchSize,1000+3*BatchSize,1000)
#"创建会话"
session=tf.Session()
session.run(tf.global_variables_initializer())
session.run(tf.local_variables_initializer())
coord=tf.train.Coordinator()
threads=tf.train.start_queue_runners(sess=session,coord=coord)
#"循环2次，从文件中随机读取"
for e in range(2):
    arrs=session.run([arrays])
    print('---第%(num)d批array---'%{'num':e+1})
    print(arrs)
coord.request_stop()
coord.join(threads)
session.close()

```

打印结果如下：

```

---"第1批array"---
[array([[[[ 1.,  2.,  3.],
           [ 4.,  5.,  6.]]],
       [[11., 12., 13.],
        [14., 15., 16.]]], dtype=float32)]
---"第2批array"---
[array([[[[21., 23., 21.],

```

```
[23., 24., 22.]],
[[[21., 23., 21.],
[23., 24., 22.]]], dtype=float32)]
```

从打印结果可以看出，每次读取两个 ndarray，并将这两个三维 ndarray 组成一个四维 ndarray 作为输出结果。以上代码中 `num_epochs=2`，代表有两轮原数据，所以可以理解为将这两轮数据随机打乱，然后每次从中读取两个 ndarray，或者说将这两轮数据分成多个批次（Batch），每次抽取一个批次，每一批中有 2 个数据，这里的 2 被称为 `BatchSize`，该操作由函数 `tf.train.shuffle_batch` 实现，以上随机读取数据的过程可以帮助我们轻松理解和实现随机梯度下降，后续会详细介绍。

至此，我们了解了如何通过 TFRecord 文件高效地管理数据，这也是利用 TensorFlow 掌握深度学习的最基础技能。

接下来，我们通过一个完整的示例介绍如何利用全连接神经网络构建分类问题的数学模型。

6.2 建立分类问题的数学模型

既然是建立分类问题的数学模型，那么我们先了解数据分类标签的数字化。

6.2.1 数据类别（标签）

我们以图 6-3 所示的数字字符的图像集为例，数字字符集一共有 10 个类别，每一幅图像都对应一个类别。



图 6-3 数字字符的图像集及其类别

例如，类别 4 可以数字化（量化）为一个一维向量来表示 [0 0 0 0 1 0 0 0 0 0]，即向量的长度是 10（有 10 个类别），因为是类别 4，所以令该向量的第 4 个（注意，这里的索引是从 0 算起的）数为 1，其他数为 0，这样量化是为了能更好地建立分类模型。TensorFlow 通过函数 `one_hot` 实现类别的数字化，示例代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
v=tf.one_hot([9,2,7,3,0,4,8,6,1,3,4,8,6,1], depth=10,axis=1,dtype=tf.float32)
session=tf.Session()
print(session.run(v))
```

打印结果如下：

```
[[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
```

以上代码中，函数 `one_hot` 中的参数 `depth=10` 代表有 10 个类别，`axis=1` 代表按照每一行存储类别的数字化，因为有 10 类，所以返回结果有 10 列。

6.2.2 图像与 TFRecord

在 6.1 节中我们已经介绍了如何把 `ndarray` 写入 `TFRecord` 文件，并从文件中解析 `ndarray`。接下来我们仍以 6.2.1 节的字符图像集为例，介绍如何利用 `TFRecord` 文件更好地管理分类问题的图像数据集（训练集）。6.2.1 节的数字字符图像集有 10 个类别，我们先创建 10 个文件夹，如图 6-4 所示。



图 6-4 存储数字字符图像的文件夹

然后在 10 个文件夹下保存对应的数字字符图像，如图 6-5 所示。

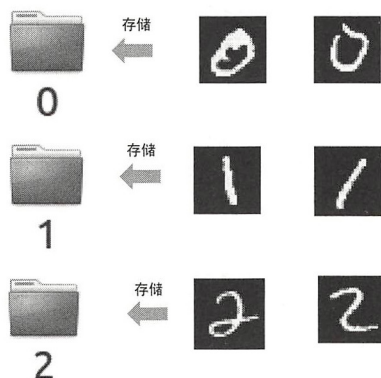


图 6-5 在每一个文件夹下存储对应的数字字符图像

其他文件夹类似，我们利用 TFRecord 文件可以更方便地管理这些数据。首先将图像数字化为 ndarray，然后利用 6.1 节介绍的将 ndarray 写入 TFRecord 的方法，分别将每一个文件夹下的数字图像保存到一个 TFRecord 文件中，具体代码如下：

```
# -*- coding: utf-8 -*-
import os
import matplotlib.image as mp_image
import tensorflow as tf
N = 10
for label in range(N):
    record=tf.python_io.TFRecordWriter(os.path.curdir+'/data/'+
                                       'data%(label)d.tfrecord'%(label))
    curDir = os.path.curdir+'/data/'+str(label)+'/'
    fileList = os.listdir(curDir)
    for name in fileList:
        #" 图片的路径和名称"
        imagePath = curDir+name
        #"读取图片，数字化为ndarray"
        image=mp_image.imread(imagePath)
        #"将ndarray二进制化"
        img_raw=image.tostring()
```

```

feature={
    'img_raw':
        tf.train.Feature(bytes_list=tf.train.BytesList(value=[img_raw])),
    'label':
        tf.train.Feature(int64_list=tf.train.Int64List(value=[label]))
}
features=tf.train.Features(feature=feature)
example=tf.train.Example(features=features)
#"字符串序列化后写入文件"
record.write(example.SerializeToString())
#"关闭文件流"
record.close()

```

运行以上程序，在文件夹中生成了 10 个文件，文件名分别为 data0.tfrecorder ~ data9.tfrecorder。每个文件存储了对应文件夹下的图像及其标签，如名称是 data4.tfrecorder 的文件存储了文件夹 ./data/4 下的每一张图片及其对应的标签。接下来，我们介绍如何从这 10 个 TFRecord 文件中，每次随机读取 3 幅图片和对应的分类标签，代码如下：

```

# -*- coding: utf-8 -*-
import os
import tensorflow as tf
import matplotlib.pyplot as plt
H,W=28,28
#"得到文件夹 ./data/ 下的所有tfRecord文件"
files=tf.train.match_filenames_once(os.path.curdir+
                                    "/data/" +
                                    "data*.tfrecord")

#"创建TFRecordReader对象"
reader=tf.TFRecordReader()
records_queue=tf.train.string_input_producer(files,num_epochs=2)
_,serialized_example=reader.read(records_queue)
#"解析文件中的图像及其对应的标签"
features=tf.parse_single_example(
    serialized_example,
    features={
        'img_raw':tf.FixedLenFeature([],tf.string),
        'label':tf.FixedLenFeature([],tf.int64),

```

```

    }

    )

#"解码二进制数据"
img_raw=features['img_raw']
img_raw=tf.decode_raw(img_raw,tf.uint8)
#"转换成图片"
img=tf.reshape(img_raw,[H,W])
#"标签"
label=features['label']
label=tf.cast(label,tf.int64)
#"每次从文件中读取3张图片"
BatchSize =3
img,label=tf.train.shuffle_batch([img,label],
    BatchSize,1000+3*BatchSize,1000)
session=tf.Session()
session.run(tf.global_variables_initializer())
session.run(tf.local_variables_initializer())
coord=tf.train.Coordinator()
threads=tf.train.start_queue_runners(sess=session,coord=coord)
print(session.run(files))
#"循环2次解析文件流中的数据"
for i in range(2):
    print('---第%(num)d批图像---'%{'num':i+1})
    imgs,labels = session.run([img,label])
    for j in range(BatchSize):
        print('-第%(num)d批图像中的第%(index)d张:标签为%(l)d-'
            %{'num':i+1,'index':j+1,'l':labels[j]})
        plt.imshow(imgs[j,:,:])
        plt.show()
    coord.request_stop()
    coord.join(threads)
    session.close()

```

运行结果如下：

```
[b'./data/data9.tfrecord' b'./data/data3.tfrecord']
```

```
b'./data/data6.tfrecord' b'./data/data5.tfrecord'
b'./data/data1.tfrecord' b'./data/data7.tfrecord'
b'./data/data4.tfrecord' b'./data/data2.tfrecord'
b'./data/data0.tfrecord' b'./data/data8.tfrecord']
```

---"第1批图像"---

- "第1批图像中的第1张: 标签为7"-



- "第1批图像中的第2张: 标签为1"-



- "第1批图像中的第3张: 标签为7"-



---"第2批图像"---

- "第2批图像中的第1张: 标签为4"-



- "第2批图像中的第2张: 标签为5"-



- "第2批图像中的第3张: 标签为6"-



以上代码中, 我们从 10 个 TFRecord 文件中循环读取了两批图像数据, 每一批有 3 幅图像, 从打印结果可以看出, 第 1 批中读取了类别分别为 7、1、7 的图像, 第 2 批中读取了类别分别为 4、5、6 的图像, 每次运行的结果可能有所不同。

至此，我们应该已经充分了解了如何通过 TFRecord 文件管理图像，以及其每一幅数字图像的标签的数字化，6.2.3 节将介绍如何利用这些已知条件，通过全连接神经网络构建分类模型。

6.2.3 建立模型

如何利用全连接神经网络建立一个图像分类的模型呢？全连接神经网络的输入层都是一维张量，而图像是高维张量（灰度图像是二维张量，彩色图像是三维张量），怎么把图像转换为一个全连接神经网络的输入呢？显然，直接将高维张量拉伸为一个一维张量即可。

以 6.2.2 节中的数字图像集为例，图像均是高为 28、宽为 28 的二维张量，我们可以建立一个输入层有 $28 \times 28 = 784$ 个神经元的全连接神经网络，其中有一个隐含层，隐含层有 200 个神经元。因为该数据集有 10 个类别，所以输出层有 10 个神经元，如图 6-6 所示。

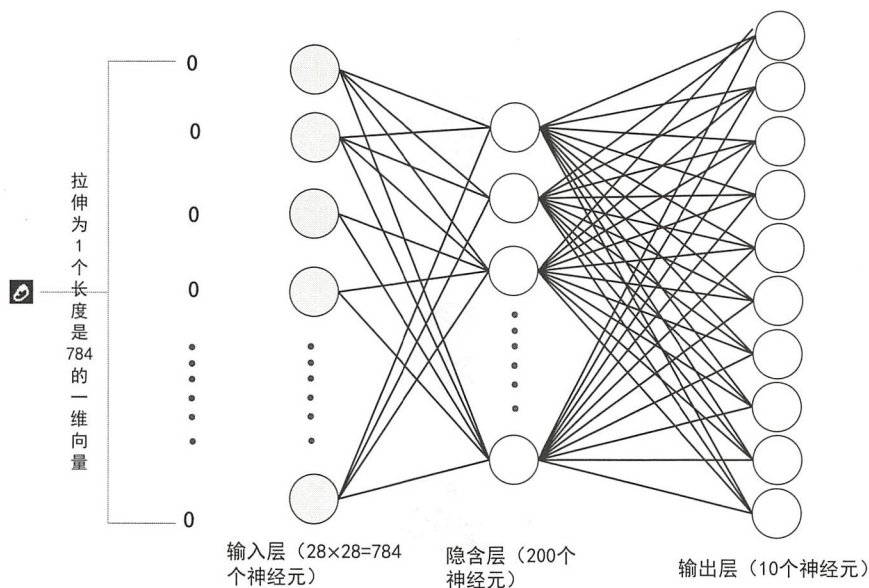


图 6-6 全连接神经网络

图像的像素值的范围为 $[0, 255]$ ，将图像拉伸为一维张量，一般不会直接输入全连接神经网络的输入层，而是将输入张量中的值标准化到 $[0, 1]$ 或者 $[-1, 1]$ 内，再输入网络的输入层。因为图像的像素值的范围为 $[0, 255]$ ，直接将张量除以 255，就可以归一到 $[0, 1]$ ，也可以将张量减去 122.5，然后除以 122.5，就归一到 $[-1, 1]$ ，所以图 6-6 不是很严谨，修改后如图 6-7 所示。

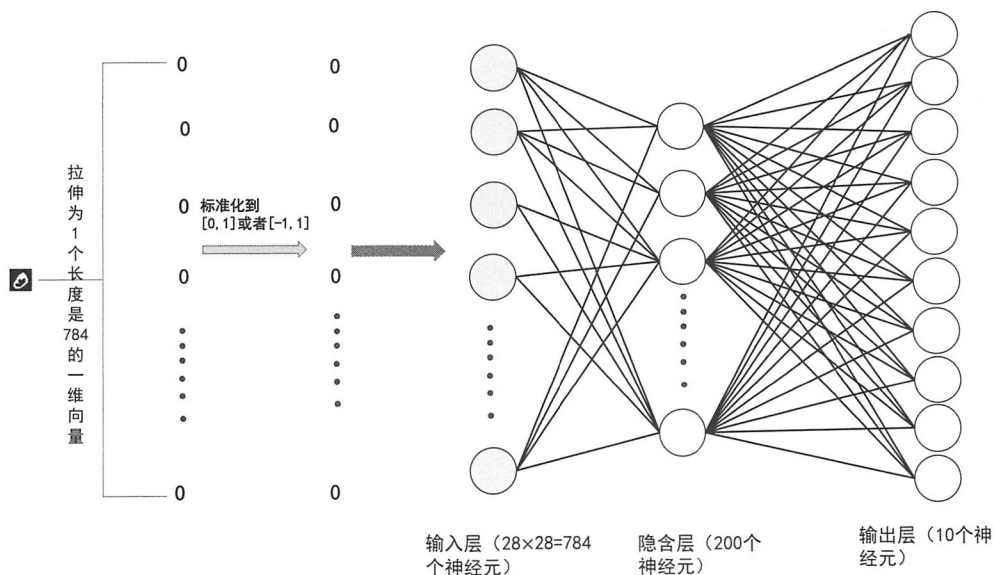


图 6-7 全连接神经网络

以上示例的实现代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
#"构建全连接神经网络"
def net(tensor):
    #"输入层、隐含层、输出层的神经元个数"
    I,H1,O=784,200,10
    #"第1层的权重矩阵和偏置"
    w1=tf.random_normal([I,H1],0,1,tf.float32)
    b1=tf.random_normal([H1],0,1,tf.float32)
    #"隐含层的结果，采用sigmoid激活函数"
    l1=tf.matmul(tensor,w1)+b1
    sigma1=tf.nn.sigmoid(l1)
    #"第2层的权重矩阵和偏置"
    w2=tf.random_normal([H1,O],0,1,tf.float32)
    b2=tf.random_normal([O],0,1,tf.float32)
    #"输出层的结果"
    l2=tf.matmul(sigma1,w2)+b2
    return l2
```



```

#"读取图片文件"
image=tf.read_file("0.jpg",'r')
#"将图片文件解码为Tensor"
image_tensor=tf.image.decode_jpeg(image)
#"图像张量的形状"
length=tf.size(image_tensor)#length=28*28
#"改变形状，拉伸为1个一维张量，按行存储"
t=tf.reshape(image_tensor,[1,length])
#"数据类型转换，转换为float32类型"
t=tf.cast(t,tf.float32)
#"标准化处理"
t=t/255.0
#"将其输入定义为2层全连接神经网络"
output=net(t)
session=tf.Session()
#打印结果：
print(session.run(output))

```

打印结果如下：

```

[[ -8.286214      0.64386976   9.21543      -0.07865417   5.6011457
   6.145635    -10.207598    7.5121603    7.7261553    9.431863  ]]

```

因为上述网络的权重矩阵和偏置都是随机生成的，所以每次运行的结果也有可能不同。我们的目标是寻找该网络的权重矩阵和偏置，使得如果输入的标签是 [1, 0, 0, 0, 0, 0, 0, 0, 0, 0] 的归一化的数据，那么输出层的结果是 [1, 0, 0, 0, 0, 0, 0, 0, 0, 0]；如果输入的标签是 [0, 1, 0, 0, 0, 0, 0, 0, 0, 0] 的归一化的数据，那么输出层的结果是 [0, 1, 0, 0, 0, 0, 0, 0, 0, 0]，其他情况类似，图 6-8 所示为类别是 0 的图像，人工分类的标签是 [1, 0, 0, 0, 0, 0, 0, 0, 0, 0]。



图 6-8 人工分类

最理想的情况是经过全连接神经网络的输出结果 \underline{y} （即全连接神经网络输出层的值，如图 6-9 所示）与人工分类结果 y 相等。

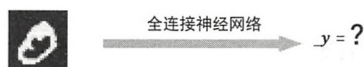


图 6-9 全连接模型分类

基本不存在一个全连接神经网络结构使得这两个结果完全吻合，但是我们可以定义一个度量来衡量 $\hat{\mathbf{y}}$ 和 \mathbf{y} 之间的差距，只要差距最小即可。如何定义一个度量来衡量 $\hat{\mathbf{y}}$ 与 \mathbf{y} 之间的差距呢？比较常用的有欧几里得距离和交叉熵，它们又称为损失值或者成本值。

对全连接神经网络处理分类问题的总结如下：在已知很多样本及其对应的标签分类的情况下，寻找一个全连接神经网络结构，包括其权重和偏置，使得所有样本经过该网络的损失值的和最小。用抽象化的数学语言来表示，即假设有 N 个样本 $\{\mathbf{x}^{(i)}\}$ ，其中 $\mathbf{x}^{(i)}$ 代表第 i 个样本， $i = 1, 2, \dots, N$ ， $\mathbf{y}^{(i)}$ 代表 $\mathbf{x}^{(i)}$ 的类别标签，用 net 代表全连接神经网络，那么 $\mathbf{x}^{(i)}$ 经过 net 后的输出层的值为 $\hat{\mathbf{y}}^{(i)} = \text{net}(\mathbf{x}^{(i)})$ ，用 $\text{loss}(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)})$ 代表第 i 个样本的人工分类结果与模型分类结果的损失值， loss 称为损失函数，我们的目标就是寻找全连接神经网络的权重和偏置，使得针对所有样本的损失值的和最小：

$$\min \sum_{i=1}^N \text{loss}(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)})$$

接下来，我们介绍常用的分类问题的损失函数。

6.3 损失函数与训练模型

对于分类问题的损失函数，常用的是 sigmoid 交叉熵和 softmax 交叉熵。我们将依次介绍它们，并介绍 TensorFlow 实现的对应函数。

6.3.1 sigmoid 损失函数

假设 \mathbf{y} 为人工分类的标签， $\hat{\mathbf{y}}$ 代表全连接神经网络的输出层的值，两者的交叉熵定义如下：

$$\begin{aligned} & -\mathbf{y} * \ln(\text{sigmoid}(\hat{\mathbf{y}})) - (1 - \mathbf{y}) * \ln(1 - \text{sigmoid}(\hat{\mathbf{y}})) \\ &= -\mathbf{y} * \ln\left(\frac{1}{1 + e^{-\hat{\mathbf{y}}}}\right) - (1 - \mathbf{y}) * \ln\left(\frac{e^{-\hat{\mathbf{y}}}}{1 + e^{-\hat{\mathbf{y}}}}\right) \\ &= \mathbf{y} * \ln(1 + e^{-\hat{\mathbf{y}}}) - [\ln(e^{-\hat{\mathbf{y}}}) - \ln(1 + e^{-\hat{\mathbf{y}}})] + \mathbf{y} * \ln(e^{-\hat{\mathbf{y}}}) - \mathbf{y} * \ln(1 + e^{-\hat{\mathbf{y}}}) \\ &= \hat{\mathbf{y}} - \hat{\mathbf{y}} * \mathbf{y} + \ln(1 + e^{-\hat{\mathbf{y}}}) \end{aligned}$$

TensorFlow 通过函数 `sigmoid_cross_entropy_with_logits(labels, logits, name)` 实现 sigmoid 交叉熵，其中 `labels` 代表人工分类标签，是一个 N 行 C 列的二维张量，代表 N 个样本的标签，每一行代表一个样本的分类标签，`logits` 代表输出层的结果，与 `labels` 的尺

寸相同，每一行代表一个样本经过全连接神经网络后的输出值，利用求和函数 `reduce_sum` 对函数 `sigmoid_cross_entropy_with_logits` 返回的结果求和，其结果为 sigmoid 交叉熵损失函数，即 `tf.reduce_sum(tf.nn.sigmoid_cross_entropy_with_logits(logits,labels))`。示例代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
#"输出层的值"
logits=tf.constant([[ -8.286214,0.64386976,9.21543,-0.07865417,5.6011457,
                      6.145635,-10.207598,7.5121603,7.7261553,9.431863]],
                    tf.float32)
#"人工分类的标签"
labels=tf.constant([[1,0,0,0,0,0,0,0,0,0]],tf.float32)
#"sigmod交叉熵"
entropy=tf.nn.sigmoid_cross_entropy_with_logits(logits=logits,labels=labels)
#"损失值"
loss=tf.reduce_sum(entropy)
#"打印损失值"
session=tf.Session()
print(session.run(loss))
```

打印结果如下：

```
55.64651
```

接下来我们介绍另一种常用的分类问题的损失函数 softmax。

6.3.2 softmax 损失函数

在介绍分类问题的 softmax 损失函数之前，我们先介绍什么是 softmax 处理。

1. softmax 计算原理

假设向量 $\mathbf{x} = (x_1, x_2, \dots, x_m)$ ，对 \mathbf{x} 进行 softmax 处理的方式如下。

$$\text{softmax}(\mathbf{x}) = \left(\frac{e^{x_1}}{\sum_{i=1}^m e^{x_i}}, \frac{e^{x_2}}{\sum_{i=1}^m e^{x_i}}, \dots, \frac{e^{x_m}}{\sum_{i=1}^m e^{x_i}} \right)$$

显然， \mathbf{x} 进行 softmax 处理后归一化为 $[0, 1]$ ，且和为 1。

举例：假设 $\mathbf{x} = (\log(2), \log(5), \log(3))$,

$$\text{softmax}(\log(2)) = \frac{\exp(\log(2))}{\exp(\log(2)) + \exp(\log(5)) + \exp(\log(3))} = \frac{2}{10} = 0.2$$

$$\text{softmax}(\log(5)) = \frac{\exp(\log(5))}{\exp(\log(2)) + \exp(\log(5)) + \exp(\log(3))} = \frac{5}{10} = 0.5$$

$$\text{softmax}(\log(3)) = \frac{\exp(\log(3))}{\exp(\log(2)) + \exp(\log(5)) + \exp(\log(3))} = \frac{3}{10} = 0.3$$

TensorFlow 通过函数 `softmax(logits,axis=None,name=None,dim=None)` 实现 softmax 处理，以上示例的代码如下：

```
import tensorflow as tf
#"输入张量"
t=tf.constant([2,5,3],tf.float32)
x=tf.log(t)
#"softmax处理"
s=tf.nn.softmax(x,0)
#"创建会话"
session=tf.Session()
#"打印结果"
print(session.run(s))
```

打印结果如下：

```
[0.19999999 0.5          0.29999998]
```

当然，函数 `tf.nn.softmax` 可以处理多维张量，示例代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
#"输入张量"
x=tf.constant([[1,2,1],
               [2,2,2]],tf.float32)
#"分别对每一行（沿"1"方向）进行softmax处理"
s=tf.nn.softmax(x,1)
#"创建会话"
session=tf.Session()
#"打印结果"
print(session.run(s))
```

打印结果如下：

```
[[0.21194157 0.5761169 0.21194157]
 [0.33333334 0.33333334 0.33333334]]
```

以上代码分别计算了二维张量的每一行的 softmax。理解了 softmax 的计算方式，接下来介绍分类问题的 softmax 损失函数。

2. softmax 熵及 softmax 损失函数

y 和 ${}_y$ 的 softmax 熵的定义如下：

$$-y * \log(\text{softmax}({}_y))$$

利用 TensorFlow 的求和函数 `reduce_sum` 和函数 `tf.nn.softmax` 定义 softmax 损失函数为 `tf.reduce_sum(-y*tf.nn.softmax({}_y,1))`。示例代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf

#"假设_y为全连接神经网络的输出（输出层有3个神经元）"
_y=tf.constant([[0,2,-3],[4,-5,6]],tf.float32)
#"人工分类结果"
y=tf.constant([[1,0,0],[0,0,1]],tf.float32)
#"softmax熵"
_y_softmax=tf.nn.softmax(_y)
entropy=tf.reduce_sum(-y*tf.log(_y_softmax),1)
#"loss损失函数"
loss=tf.reduce_sum(entropy)
#"创建会话"
session=tf.Session()
#"打印结果"
print(session.run(loss))
```

打印结果如下：

```
2.259788
```

以上代码实现中，使用了乘法运算符“*”和函数 `tf.nn.softmax` 实现 softmax 熵，TensorFlow 将这两个运算封装成了一个函数 `softmax_cross_entropy_with_logits_v2`，利用该函数可以直接得到 softmax 熵，即

```
tf.reduce_sum(tf.nn.softmax_cross_entropy_with_logits_v2(logits,labels))
```

所以，以上示例代码也可以用以下代码完成：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
#"假设_y为全连接神经网络的输出(输出层有3个神经元)"
_y=tf.constant([[0,2,-3],[4,-5,6]],tf.float32)
#"人工分类结果"
y=tf.constant([[1,0,0],[0,0,1]],tf.float32)
#"softmax熵"
entropy=tf.nn.softmax_cross_entropy_with_logits_v2(logits=_y,labels=y)
#"loss损失函数"
loss=tf.reduce_sum(entropy)
#"创建会话"
session=tf.Session()
#"打印结果"
print(session.run(loss))
```

打印结果如下：

2.259788

在图 6-7 的基础上加上 softmax 处理，如图 6-10 所示，即可表示利用 softmax 损失函数处理分类问题的全连接神经网络。

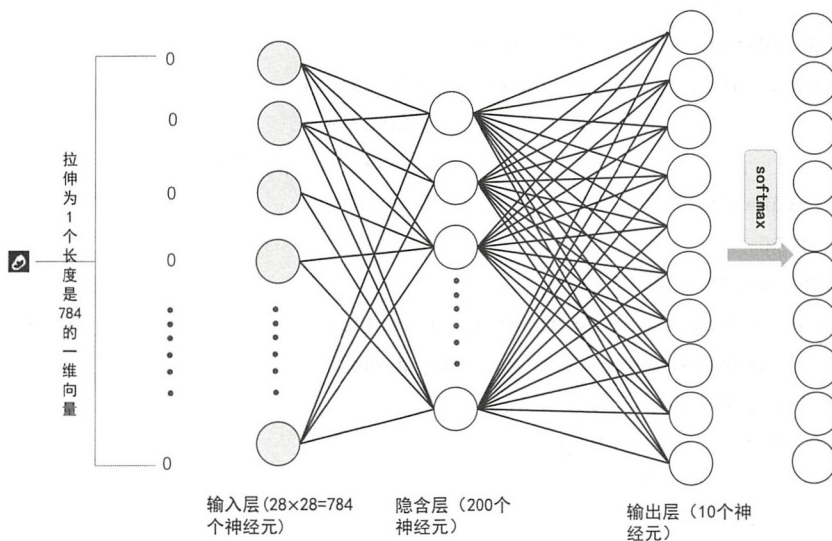


图 6-10 softmax 处理的全连接神经网络



至此，我们已经了解了常用的分类问题的损失函数，接下来就可以利用第 3 章介绍的各种梯度下降法求解损失函数最小值的最优解，即训练模型。

6.3.3 训练和评估模型

直到现在我们已经掌握了：

- (1) 从 TFRecord 中解析数据。
- (2) 构建全连接神经网络。
- (3) 构造损失函数。
- (4) 梯度下降法。

将这 4 部分组合起来就可以训练一个分类问题的全连接神经网络模型，处理 6.2 节的数字图像集的分类问题的代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
import os
#"占位符"
x=tf.placeholder(tf.float32,[None,28*28])
labels=tf.placeholder(tf.float32,[None,10])
#"-----第1步：解析数据-----"
nums=33#"所有训练样本的个数"
#"得到文件夹./data/下的所有tfRecord文件"
files=tf.train.match_filenames_once(os.path.curdir+
                                   "/data/"+
                                   "data*.tfrecord")

#"创建TFRecordReader对象"
num_epochs=1000
reader=tf.TFRecordReader()
records_queue=tf.train.string_input_producer(files,num_epochs=num_epochs)
_,serialized_example=reader.read(records_queue)
#"解析文件中的图像及其对应的标签"
features=tf.parse_single_example(
    serialized_example,
    features={
        'img_raw':tf.FixedLenFeature([],tf.string),
```



```

        'label':tf.FixedLenFeature([],tf.int64),
    }

    )

#"解码二进制数据"
img_raw=features['img_raw']
img_raw=tf.decode_raw(img_raw,tf.uint8)
img=tf.reshape(img_raw,[28*28])
img=tf.cast(img,tf.float32)
img=img/255.0
#"标签"
label=features['label']
label=tf.cast(label,tf.int64)
label_onehot=tf.one_hot(label,10,dtype=tf.float32)
#"每次从文件中读取3张图片"
BatchSize =3
imgs,labels_onehot=tf.train.shuffle_batch([img,label_onehot],
                                           BatchSize,1000+3*BatchSize,1000)

#"-----第2部分：构建全连接神经网络-----"
#"输入层、隐含层、输出层的神经元个数"
I,H1,0=784,200,10
#"输入层到隐含层的权重矩阵和偏置"
w1=tf.Variable(tf.random_normal([I,H1],0,1,tf.float32),
               dtype=tf.float32,name='w1')
b1=tf.Variable(tf.random_normal([H1],0,1,tf.float32),
               dtype=tf.float32,name='b1')
#"隐含层的结果，采用sigmoid激活函数"
l1=tf.matmul(x,w1)+b1
sigma1=tf.nn.sigmoid(l1)
#"第2层隐含层到输出层的权重矩阵和偏置"
w2=tf.Variable(tf.random_normal([H1,0],0,1,tf.float32),
               dtype=tf.float32,name='w2')
b2=tf.Variable(tf.random_normal([0],0,1,tf.float32),
               dtype=tf.float32,name='b2')
#"输出层的结果"
logits=tf.matmul(sigma1,w2)+b2

```



```

#-----第3部分：构造损失函数-----"
loss=tf.reduce_sum(tf.nn.softmax_cross_entropy_with_logits_v2(
                                labels=labels_onehot,logits=logits))
#-----第4部分：梯度下降-----"
opti=tf.train.AdamOptimizer(0.001,0.9,0.999,1e-8).minimize(loss)
#"创建会话"
session=tf.Session()
session.run(tf.global_variables_initializer())
session.run(tf.local_variables_initializer())
coord=tf.train.Coordinator()
threads=tf.train.start_queue_runners(sess=session,coord=coord)
for i in range(num_epochs):
    for n in range(int(nums/BatchSize)):
        imgs_arr,labes_onehot_arr=session.run([imgs,labels_onehot])
        session.run(opti,feed_dict={x:imgs_arr,labels:labes_onehot_arr})
coord.request_stop()
coord.join(threads)
session.close()

```

以上代码中函数 `tf.train.shuffle_batch` 和 `tf.train.AdamOptimizer` 的组合使用其实就实现了 3.3.9 节介绍的用随机梯度下降法处理函数的最小值。通过上述代码，我们计算出了权重矩阵和偏置。接下来，我们选取一幅图片（当然需要进行归一化处理）作为全连接神经网络的输入，然后将得到的网络输出值经过 softmax 处理，显然，一般不会得到类似以下理想中的值：

$$[0, 1, 0, 0, 0, 0, 0, 0, 0]$$

如果得到以下值：

$$[0.1, 0.3, 0.1, 0.1, 0, 0.1, 0.05, 0.04, 0.01]$$

应该归为哪一类呢？因为 0.3 是最大的，它的位置索引是 1，所以属于第 1 类。上述代码只是计算模型参数（即全连接神经网络的权重和偏置），如何评估模型的参数呢？需要加入计算准确率的代码，如下：

```

pred_correct=tf.equal(tf.argmax(tf.nn.softmax(logits,1)),
                        tf.argmax(labels,1))
accuracy=tf.reduce_mean(tf.cast(pred_correct),tf.float)

```



其中函数 `equal`、`argmax`、`reduce_mean`、`cast` 在第 2 章中都有详细介绍，本节不再赘述。

假设向量 $\mathbf{x} = (x_1, x_2, \dots, x_m)$ 经过 softmax 处理后的结果为

$$\text{softmax}(\mathbf{x}) = \left(\frac{e^{x_1}}{\sum_{i=1}^m e^{x_i}}, \frac{e^{x_2}}{\sum_{i=1}^m e^{x_i}}, \dots, \frac{e^{x_m}}{\sum_{i=1}^m e^{x_i}} \right)$$

因为如果 $x_p < x_q$ ，那么 $\frac{e^{x_p}}{\sum_{i=1}^m e^{x_i}} < \frac{e^{x_q}}{\sum_{i=1}^m e^{x_i}}$ ，所以 `tf.argmax(tf.nn.softmax(logits,1))` 等价于 `tf.argmax(logits,1)`，上述评估模型正确率的代码可以写为

```
pred_correct=tf.equal(tf.argmax(logits,1),tf.argmax(labels,1))
accuracy=tf.reduce_mean(tf.cast(pred_correct),tf.float)
```

至此，我们完整地介绍了如何通过 TensorFlow 构建一个全连接神经网络处理分类问题。

既然介绍神经网络，有一个问题是避不开的，那就是神经网络的梯度反向传播算法（常称为 BP 算法），该算法的提出主要是解决快速计算梯度的问题。我们来分析利用图 6-10 所示的全连接神经网络解决数字字符的分类问题时构造的损失函数的自变量个数，即该网络的权重和偏置的个数。从第 0 层到第 1 层的权重和偏置个数为 $784 \times 200 + 200 = 157000$ ，从第 1 层到第 2 层的权重和偏置个数为 $200 \times 10 + 10 = 2010$ ，所以损失函数的自变量个数为 $157000 + 2010 = 159010$ 。如果增加网络层数及每一层的神经元个数，那么损失函数的自变量个数会更多，在利用梯度下降法时，需要计算每个自变量的偏导数，这个计算量和复杂度是非常大的，而神经网络的梯度反向传播算法巧妙地解决了该问题，6.4 节将详细介绍该算法。

6.4 全连接神经网络的梯度反向传播

6.4.1 数学原理及示例

我们通过以下示例来理解梯度反向传播。假设全连接神经网络如图 6-11 所示，其中有 2 个隐含层，输入层的神经个数为 2，第 1 层的神经元个数为 3，第 2 层的神经元个数为 3，输出层的神经元个数为 2，激活函数为 $f(\cdot)$ ，第 1 层和第 2 层有激活函数操作，输出层没有激活函数操作，权重和偏置均为未知数。假设该全连接神经网络有一个输入 $[2, 3]$ ，根据前馈全连接神经网络的计算方式，可以计算出每一层的线性组合及其对应的激活值，如图 6-12 所示。为了便于讨论，我们将分别显示中间的两个隐含层计算出来的线性组合和激活值。



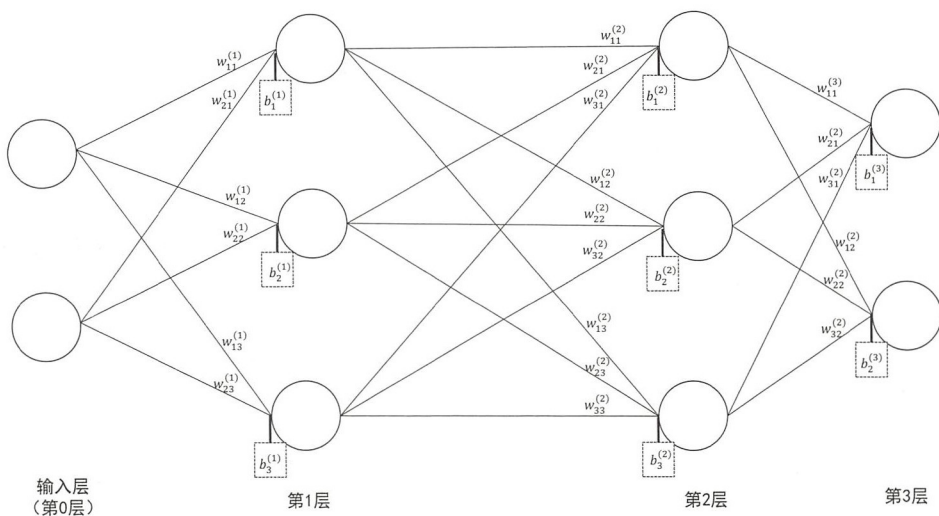


图 6-11 全连接神经网络

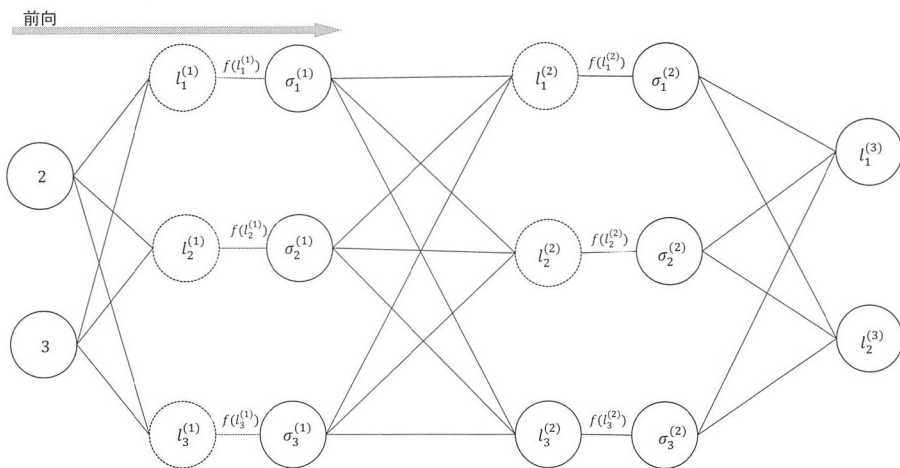


图 6-12 根据输入值计算每一层的线性组合及其对应的激活值

假设函数 F 是根据输出层的值 $l_1^{(3)}$ 、 $l_2^{(3)}$ 构造出的函数，那么函数 F 就是以全连接神经网络权重和偏置为自变量的函数，其中自变量的个数为 29，假设为

$$F(w_{11}^{(1)}, w_{21}^{(1)}, b_1^{(1)}, w_{12}^{(1)}, w_{22}^{(1)}, b_2^{(1)}, w_{13}^{(1)}, w_{23}^{(1)}, b_3^{(1)}, w_{11}^{(2)}, w_{21}^{(2)}, w_{31}^{(2)}, b_1^{(2)}, w_{12}^{(2)}, w_{22}^{(2)}, w_{32}^{(2)}, b_2^{(2)}, w_{13}^{(2)}, w_{23}^{(2)}, w_{33}^{(2)}, b_3^{(2)}, w_{11}^{(3)}, w_{21}^{(3)}, w_{31}^{(3)}, b_1^{(3)}, w_{12}^{(3)}, w_{22}^{(3)}, w_{32}^{(3)}, b_2^{(3)}) = (l_1^{(3)} + l_2^{(3)})^2$$



$l_1^{(3)}$ 、 $l_2^{(3)}$ 是关于 $\sigma_1^{(2)}$ 、 $\sigma_2^{(2)}$ 、 $\sigma_3^{(2)}$ 的函数，根据导数的链式法则， F 关于 $\sigma_1^{(2)}$ 的导数为

$$\frac{\partial F}{\partial \sigma_1^{(2)}} = \frac{\partial F}{\partial l_1^{(3)}} \frac{\partial l_1^{(3)}}{\partial \sigma_1^{(2)}} + \frac{\partial F}{\partial l_2^{(3)}} \frac{\partial l_2^{(3)}}{\partial \sigma_1^{(2)}} = \frac{\partial F}{\partial l_1^{(3)}} w_{11}^{(3)} + \frac{\partial F}{\partial l_2^{(3)}} w_{12}^{(3)}$$

F 关于 $\sigma_2^{(2)}$ 的导数为

$$\frac{\partial F}{\partial \sigma_2^{(2)}} = \frac{\partial F}{\partial l_1^{(3)}} \frac{\partial l_1^{(3)}}{\partial \sigma_2^{(2)}} + \frac{\partial F}{\partial l_2^{(3)}} \frac{\partial l_2^{(3)}}{\partial \sigma_2^{(2)}} = \frac{\partial F}{\partial l_2^{(3)}} w_{21}^{(3)} + \frac{\partial F}{\partial l_2^{(3)}} w_{22}^{(3)}$$

F 关于 $\sigma_3^{(2)}$ 的导数为

$$\frac{\partial F}{\partial \sigma_3^{(2)}} = \frac{\partial F}{\partial l_1^{(3)}} \frac{\partial l_1^{(3)}}{\partial \sigma_3^{(2)}} + \frac{\partial F}{\partial l_2^{(3)}} \frac{\partial l_2^{(3)}}{\partial \sigma_3^{(2)}} = \frac{\partial F}{\partial l_1^{(3)}} w_{31}^{(3)} + \frac{\partial F}{\partial l_2^{(3)}} w_{32}^{(3)}$$

仔细观察后会发现，上述过程可以用一个全连接神经网络表示，如图 6-13 所示。

因为 $\sigma_1^{(2)} = f(l_1^{(2)})$ ，所以

$$\frac{\partial F}{\partial l_1^{(2)}} = \frac{\partial F}{\partial \sigma_1^{(2)}} f'(l_1^{(2)})$$

在图 6-12 的基础上进行修改，得到图 6-14。

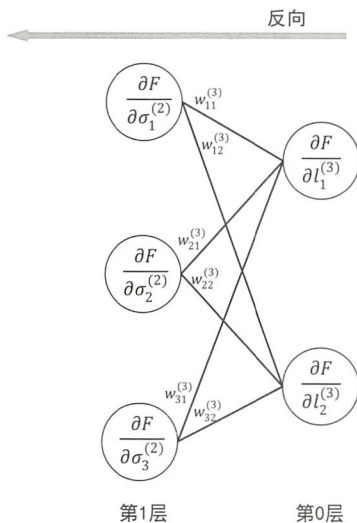


图 6-13 $\frac{\partial F}{\partial l_1^{(3)}}$ 与 $\frac{\partial F}{\partial \sigma_1^{(2)}}$ 的关系

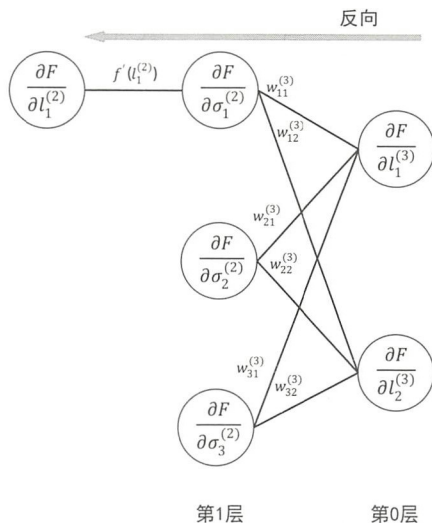


图 6-14 $\frac{\partial F}{\partial l_1^{(2)}}$ 与 $\frac{\partial F}{\partial \sigma_1^{(2)}}$ 的关系

显然，图 6-14 不符合全连接神经网络的定义，因为全连接神经网络上一层的神经元都



有下一层连接，所以我们可以做以下（补零操作）处理：

$$\frac{\partial F}{\partial l_1^{(2)}} = f'(l_1^{(2)}) \cdot \frac{\partial F}{\partial \sigma_1^{(2)}} + 0 \cdot \frac{\partial F}{\partial \sigma_2^{(2)}} + 0 \cdot \frac{\partial F}{\partial \sigma_3^{(2)}}$$

则图 6-14 可以修改为如图 6-15 所示的全连接神经网络。同理，可以得到

$$\frac{\partial F}{\partial l_2^{(2)}} = 0 \cdot \frac{\partial F}{\partial \sigma_1^{(2)}} + f'(l_2^{(2)}) \cdot \frac{\partial F}{\partial \sigma_2^{(2)}} + 0 \cdot \frac{\partial F}{\partial \sigma_3^{(2)}}$$

$$\frac{\partial F}{\partial l_3^{(2)}} = 0 \cdot \frac{\partial F}{\partial \sigma_1^{(2)}} + 0 \cdot \frac{\partial F}{\partial \sigma_2^{(2)}} + f'(l_3^{(2)}) \cdot \frac{\partial F}{\partial \sigma_3^{(2)}}$$

同理，上述关系可以在图 6-15 的基础上修改为如图 6-16 所示。

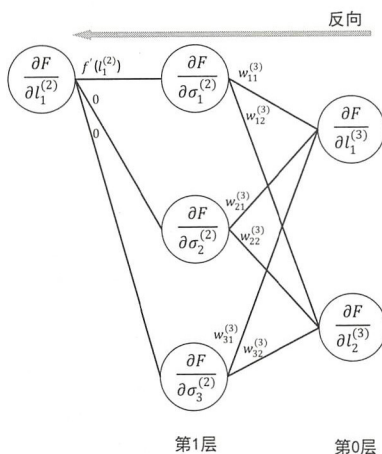


图 6-15 $\frac{\partial F}{\partial l_1^{(2)}}$ 与 $\frac{\partial F}{\partial \sigma_1^{(2)}}$ 的关系

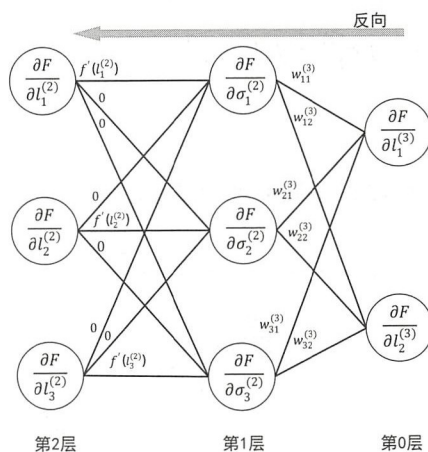


图 6-16 $\frac{\partial F}{\partial l_1^{(2)}}$ 与 $\frac{\partial F}{\partial \sigma_1^{(2)}}$ 的关系

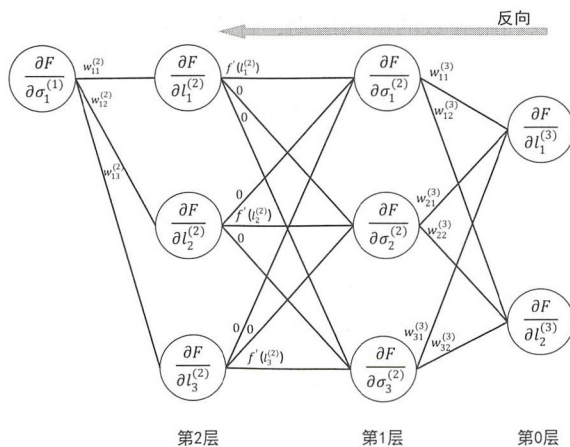
计算出了 $\frac{\partial F}{\partial l_1^{(2)}}$ 、 $\frac{\partial F}{\partial l_2^{(2)}}$ 、 $\frac{\partial F}{\partial l_3^{(2)}}$ ，又因为 $l_1^{(2)}$ 、 $l_2^{(2)}$ 、 $l_3^{(2)}$ 是关于 $\sigma_1^{(1)}$ 的函数，根据链式法则计算

$\frac{\partial F}{\partial \sigma_1^{(1)}}$ 为

$$\frac{\partial F}{\partial \sigma_1^{(1)}} = \frac{\partial F}{\partial l_1^{(2)}} \frac{\partial l_1^{(2)}}{\partial \sigma_1^{(1)}} + \frac{\partial F}{\partial l_2^{(2)}} \frac{\partial l_2^{(2)}}{\partial \sigma_1^{(1)}} + \frac{\partial F}{\partial l_3^{(2)}} \frac{\partial l_3^{(2)}}{\partial \sigma_1^{(1)}} = \frac{\partial F}{\partial l_1^{(2)}} w_{11}^{(2)} + \frac{\partial F}{\partial l_2^{(2)}} w_{12}^{(2)} + \frac{\partial F}{\partial l_3^{(2)}} w_{13}^{(2)}$$

同理，上述关系可以在图 6-16 的基础上修改为如图 6-17 所示。



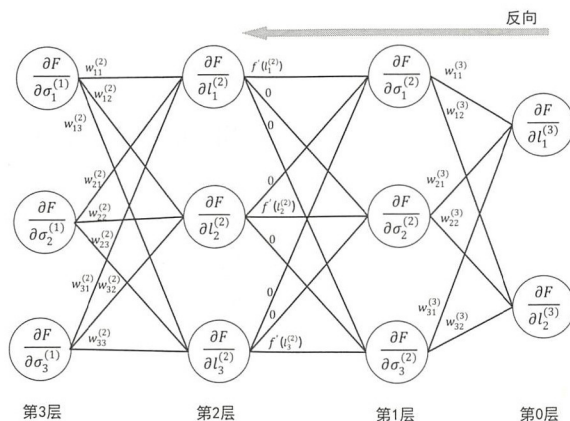
图 6-17 $\frac{\partial F}{\partial \sigma_1^{(1)}}$ 与 $\frac{\partial F}{\partial l^{(2)}}$ 的关系

同理，根据链式法则计算 $\frac{\partial F}{\partial \sigma_2^{(1)}}$ 、 $\frac{\partial F}{\partial \sigma_3^{(1)}}$ ，计算过程如下：

$$\frac{\partial F}{\partial \sigma_2^{(1)}} = \frac{\partial F}{\partial l_1^{(2)}} \frac{\partial l_1^{(2)}}{\partial \sigma_2^{(1)}} + \frac{\partial F}{\partial l_2^{(2)}} \frac{\partial l_2^{(2)}}{\partial \sigma_2^{(1)}} + \frac{\partial F}{\partial l_3^{(2)}} \frac{\partial l_3^{(2)}}{\partial \sigma_2^{(1)}} = \frac{\partial F}{\partial l_1^{(2)}} w_{21}^{(2)} + \frac{\partial F}{\partial l_2^{(2)}} w_{22}^{(2)} + \frac{\partial F}{\partial l_3^{(2)}} w_{23}^{(2)}$$

$$\frac{\partial F}{\partial \sigma_3^{(1)}} = \frac{\partial F}{\partial l_1^{(2)}} \frac{\partial l_1^{(2)}}{\partial \sigma_3^{(1)}} + \frac{\partial F}{\partial l_2^{(2)}} \frac{\partial l_2^{(2)}}{\partial \sigma_3^{(1)}} + \frac{\partial F}{\partial l_3^{(2)}} \frac{\partial l_3^{(2)}}{\partial \sigma_3^{(1)}} = \frac{\partial F}{\partial l_1^{(2)}} w_{31}^{(2)} + \frac{\partial F}{\partial l_2^{(2)}} w_{32}^{(2)} + \frac{\partial F}{\partial l_3^{(2)}} w_{33}^{(2)}$$

上述关系可以添加到图 6-17 的全连接神经网络中，得到如图 6-18 所示的新的全连接神经网络。

图 6-18 $\frac{\partial F}{\partial \sigma_1^{(1)}}$ 与 $\frac{\partial F}{\partial l^{(2)}}$ 的关系

依此类推：我们可以利用导数的链式法则，计算出 $\frac{\partial F}{\partial l^{(1)}}$ 与 $\frac{\partial F}{\partial \sigma^{(1)}}$ 的关系，可以在图 6-18 的基础上加上一层表示该关系，这样就得到了 F 关于中间变量的偏导数，而这些偏导数的关系也可以用一个全连接神经网络表示，如图 6-19 所示。

如何利用上述结论，快速计算 F 在某一特定点的梯度呢？假设要计算 F 在点

$$(w_{11}^{(1)}, w_{21}^{(1)}, b_1^{(1)}, w_{12}^{(1)}, w_{22}^{(1)}, b_2^{(1)}, w_{13}^{(1)}, w_{23}^{(1)}, b_3^{(1)}, w_{11}^{(2)}, w_{21}^{(2)}, w_{31}^{(2)}, b_1^{(2)}, w_{12}^{(2)}, w_{22}^{(2)}, w_{32}^{(2)}, b_2^{(2)}, w_{13}^{(2)}, w_{23}^{(2)}, w_{33}^{(2)}, b_3^{(2)}, w_{11}^{(3)}, w_{21}^{(3)}, w_{31}^{(3)}, b_1^{(3)}, w_{12}^{(3)}, w_{22}^{(3)}, w_{32}^{(3)}, b_2^{(3)}) =$$

$$(1, 2, 3, 4, 5, 6, 7, 8, 9, 9, 6, 5, 4, 7, 4, 2, 1, 3, 5, 8, 4, 1, 6, 8, 2, 4, 8, 1, 3)$$

处的梯度，为了表示方便将该点记为 \mathbf{wb} ，只要计算 F 在该点处每一个自变量的偏导数即可，比如 F 在该点处关于 $b_1^{(3)}$ 的偏导数，记为

$$\frac{\partial F}{\partial b_1^{(3)}} \bigg|_{\mathbf{wb}, b_1^{(3)}=9}$$

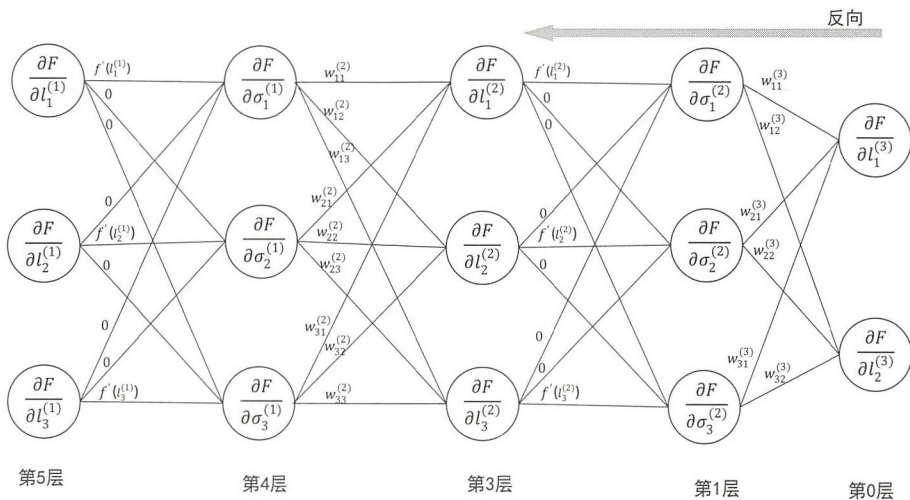


图 6-19 F 关于中间变量的偏导数的链式法则

根据导数的链式法则

$$\frac{\partial F}{\partial b_1^{(3)}} = \frac{\partial F}{\partial l_1^{(3)}} \frac{\partial l_1^{(3)}}{\partial b_1^{(3)}}$$

因为

$$\frac{\partial l_1^{(3)}}{\partial b_1^{(3)}} = 1$$



所以

$$\frac{\partial F}{\partial b_1^{(3)}} = \frac{\partial F}{\partial l_1^{(3)}}$$

针对其他偏置的偏导数也类似，为了方便，我们用矩阵表示为

$$\frac{\partial F}{\partial \mathbf{b}^{(3)}} = \begin{bmatrix} \frac{\partial F}{\partial b_1^{(3)}} \\ \frac{\partial F}{\partial b_2^{(3)}} \end{bmatrix} = \begin{bmatrix} \frac{\partial F}{\partial l_1^{(3)}} \\ \frac{\partial F}{\partial l_2^{(3)}} \end{bmatrix}$$

首先根据前馈网络的计算方法，计算当权重和偏置为 \mathbf{wb} 时，关于 F 的中间变量的值，即每一层的线性组合及相应的激活后的值，如图 6-20 所示。

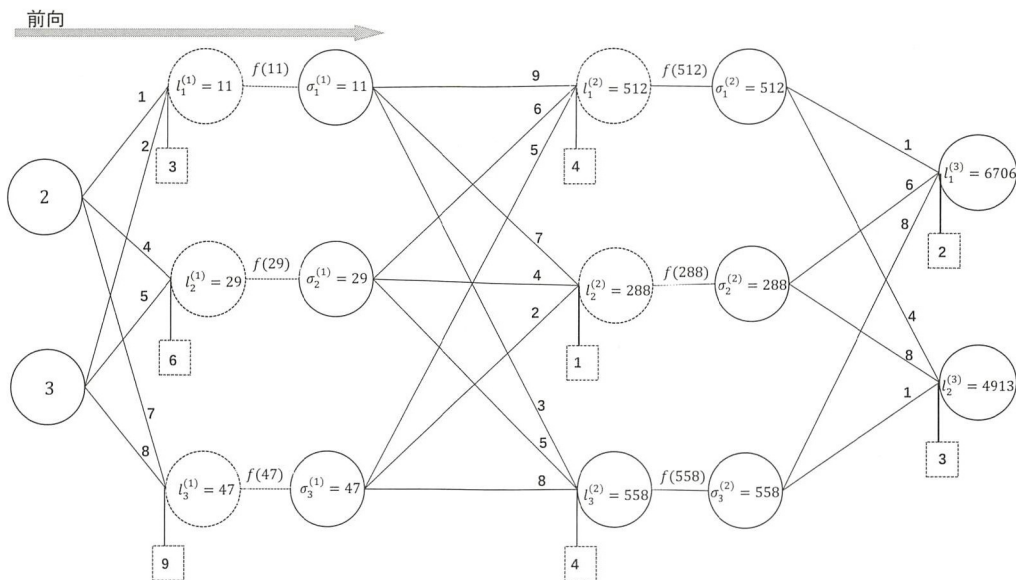


图 6-20 当权重和偏置为 \mathbf{wb} 时，全连接神经网络每一层的线性组合及相应的激活值

因为

$$\frac{\partial F}{\partial \mathbf{b}^{(3)}} = \frac{\partial F}{\partial \mathbf{l}^{(3)}}$$

所以



$$\frac{\partial F}{\partial \mathbf{b}^{(3)}}|_{\mathbf{wb}, \mathbf{b}_1^{(3)}=\begin{bmatrix} 2 \\ 3 \end{bmatrix}} = \begin{bmatrix} \frac{\partial F}{\partial \mathbf{b}^{(3)}}|_{\mathbf{wb}, \mathbf{b}_1^{(3)}=2} \\ \frac{\partial F}{\partial \mathbf{b}^{(3)}}|_{\mathbf{wb}, \mathbf{b}_2^{(3)}=3} \end{bmatrix} = \frac{\partial F}{\partial \mathbf{l}^{(3)}}|_{\mathbf{l}^{(3)}=\begin{bmatrix} 6706 \\ 4913 \end{bmatrix}} = \begin{bmatrix} \frac{\partial F}{\partial \mathbf{l}^{(3)}}|_{\mathbf{l}_1^{(3)}=6706} \\ \frac{\partial F}{\partial \mathbf{l}^{(3)}}|_{\mathbf{l}_2^{(3)}=4913} \end{bmatrix}$$

因为

$$\frac{\partial F}{\partial \mathbf{l}_1^{(3)}}|_{\mathbf{l}_1^{(3)}=6706} = 2 \times (6706 + 4713) = 23238, \quad \frac{\partial F}{\partial \mathbf{l}_2^{(3)}}|_{\mathbf{l}_2^{(3)}=4913} = 2 \times (6706 + 4713) = 23238$$

所以

$$\frac{\partial F}{\partial \mathbf{b}^{(3)}}|_{\mathbf{wb}, \mathbf{b}^{(3)}=\begin{bmatrix} 2 \\ 3 \end{bmatrix}} = \begin{bmatrix} 23238 \\ 23238 \end{bmatrix}$$

现在已经计算出了 $\frac{\partial F}{\partial \mathbf{l}_1^{(3)}}|_{\mathbf{l}_1^{(3)}=6706}$ 和 $\frac{\partial F}{\partial \mathbf{l}_2^{(3)}}|_{\mathbf{l}_2^{(3)}=4913}$ ，相当于已知了图 6-19 所示的全连接神经网络的输入层，我们结合图 6-20 就可以快速地计算出 F 在

$$\boldsymbol{\sigma}^{(2)} = \begin{bmatrix} \sigma_1^{(2)} \\ \sigma_2^{(2)} \\ \sigma_3^{(2)} \end{bmatrix} = \begin{bmatrix} 512 \\ 288 \\ 558 \end{bmatrix}$$

处的梯度为 $\frac{\partial F}{\partial \boldsymbol{\sigma}^{(2)}}$ ，然后计算 F 在

$$\mathbf{l}^{(2)} = \begin{bmatrix} l_1^{(2)} \\ l_2^{(2)} \\ l_3^{(2)} \end{bmatrix} = \begin{bmatrix} 512 \\ 288 \\ 558 \end{bmatrix}$$

处的梯度为 $\frac{\partial F}{\partial \mathbf{l}^{(2)}}$ ，然后计算 F 在

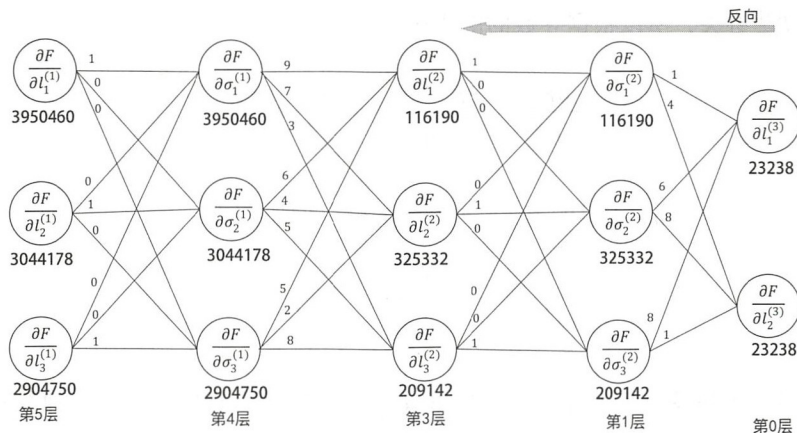
$$\boldsymbol{\sigma}^{(1)} = \begin{bmatrix} \sigma_1^{(1)} \\ \sigma_2^{(1)} \\ \sigma_3^{(1)} \end{bmatrix} = \begin{bmatrix} 11 \\ 29 \\ 47 \end{bmatrix}$$

处的梯度为 $\frac{\partial F}{\partial \boldsymbol{\sigma}^{(1)}}$ ，然后计算 F 在

$$\mathbf{l}^{(1)} = \begin{bmatrix} l_1^{(1)} \\ l_2^{(1)} \\ l_3^{(1)} \end{bmatrix} = \begin{bmatrix} 11 \\ 29 \\ 47 \end{bmatrix}$$

处的梯度为 $\frac{\partial F}{\partial \mathbf{l}^{(1)}}$ ，结果如图 6-21 所示。



图 6-21 函数 F 关于中间变量的梯度

因为根据导数的链式法则，函数 F 关于第 2 层和第 1 层的偏置的偏导数满足以下关系：

$$\frac{\partial F}{\partial \mathbf{b}^{(2)}} = \begin{bmatrix} \frac{\partial F}{\partial b_1^{(2)}} \\ \frac{\partial F}{\partial b_2^{(2)}} \\ \frac{\partial F}{\partial b_3^{(2)}} \end{bmatrix} = \begin{bmatrix} \frac{\partial F}{\partial l_1^{(2)}} \\ \frac{\partial F}{\partial l_2^{(2)}} \\ \frac{\partial F}{\partial l_3^{(2)}} \end{bmatrix}, \quad \frac{\partial F}{\partial \mathbf{b}^{(1)}} = \begin{bmatrix} \frac{\partial F}{\partial b_1^{(1)}} \\ \frac{\partial F}{\partial b_2^{(1)}} \\ \frac{\partial F}{\partial b_3^{(1)}} \end{bmatrix} = \begin{bmatrix} \frac{\partial F}{\partial l_1^{(1)}} \\ \frac{\partial F}{\partial l_2^{(1)}} \\ \frac{\partial F}{\partial l_3^{(1)}} \end{bmatrix}$$

所以如图 6-21 所示，我们可以快速地得到 F 在点 \mathbf{wb} 处关于 $\mathbf{b}^{(2)}$ 和 $\mathbf{b}^{(1)}$ 的梯度，结果为

$$\begin{aligned} \frac{\partial F}{\partial \mathbf{b}^{(2)}} \Big|_{\mathbf{wb}, \mathbf{b}^{(2)} = \begin{bmatrix} 4 \\ 1 \\ 4 \end{bmatrix}} &= \begin{bmatrix} \frac{\partial F}{\partial b_1^{(2)}} \Big|_{\mathbf{wb}, b_1^{(2)}=4} \\ \frac{\partial F}{\partial b_2^{(2)}} \Big|_{\mathbf{wb}, b_2^{(2)}=1} \\ \frac{\partial F}{\partial b_3^{(2)}} \Big|_{\mathbf{wb}, b_3^{(2)}=4} \end{bmatrix} = \begin{bmatrix} \frac{\partial F}{\partial l_1^{(2)}} \Big|_{l_1^{(2)}=512} \\ \frac{\partial F}{\partial l_2^{(2)}} \Big|_{l_2^{(2)}=298} \\ \frac{\partial F}{\partial l_3^{(2)}} \Big|_{l_3^{(2)}=558} \end{bmatrix} = \begin{bmatrix} 116190 \\ 325332 \\ 209142 \end{bmatrix} \\ \\ \frac{\partial F}{\partial \mathbf{b}^{(1)}} \Big|_{\mathbf{wb}, \mathbf{b}^{(1)} = \begin{bmatrix} 3 \\ 6 \\ 9 \end{bmatrix}} &= \begin{bmatrix} \frac{\partial F}{\partial b_1^{(1)}} \Big|_{\mathbf{wb}, b_1^{(1)}=3} \\ \frac{\partial F}{\partial b_2^{(1)}} \Big|_{\mathbf{wb}, b_2^{(1)}=6} \\ \frac{\partial F}{\partial b_3^{(1)}} \Big|_{\mathbf{wb}, b_3^{(1)}=9} \end{bmatrix} = \begin{bmatrix} \frac{\partial F}{\partial l_1^{(3)}} \Big|_{l_1^{(3)}=11} \\ \frac{\partial F}{\partial l_2^{(3)}} \Big|_{l_2^{(3)}=29} \\ \frac{\partial F}{\partial l_3^{(3)}} \Big|_{l_3^{(3)}=47} \end{bmatrix} = \begin{bmatrix} 3950460 \\ 3044178 \\ 2904750 \end{bmatrix} \end{aligned}$$

我们已经计算出了函数 F 关于偏置的偏导数，接着介绍 F 关于权重 $w_{ij}^{(n)}$ 的偏导数，因

为 $w_{ij}^{(n)}$ 只与 $l_j^{(n)}$ 有关，所以根据导数的链式法则得到

$$\frac{\partial F}{\partial w_{ij}^{(n)}} = \frac{\partial F}{\partial l_j^{(n)}} \frac{\partial l_j^{(n)}}{\partial w_{ij}^{(n)}} = \frac{\partial F}{\partial l_j^{(n)}} \sigma_i^{(n-1)}$$

其中 $\sigma_i^{(n-1)}$ 代表第 $n-1$ 层的第 i 个神经元的值，例如：

$$\frac{\partial F}{\partial w_{11}^{(2)}} = \frac{\partial F}{\partial l_1^{(2)}} \frac{\partial l_1^{(2)}}{\partial w_{11}^{(2)}} = \frac{\partial F}{\partial l_1^{(2)}} \sigma_1^{(1)}$$

以下从另一个角度解释上述公式，权重 $w_{11}^{(1)}$ 连接的是第 0 层的第 1 个神经元到第 1 层的第 1 个神经元，在计算 F 关于 $w_{11}^{(1)}$ 的偏导数时，只要找到图 6-11 中 $\sigma_1^{(0)}$ 的值，该值为输入层的第 1 个神经元处的值，再根据图 6-19 找到 $\frac{\partial F}{\partial l_1^{(1)}}$ 的值，然后将其相乘就可以了，其他的权重的偏导数类似。

根据上述结论，假设要计算 F 在点 \mathbf{wb} 处关于 $w_{11}^{(1)}$ 的偏导数 $\frac{\partial F}{\partial w_{11}^{(1)}} |_{\mathbf{wb}, w_{11}^{(1)}=1}$ ，首先根据图 6-20 找到第 0 层的第 1 个神经元的值，如图 6-22 所示。

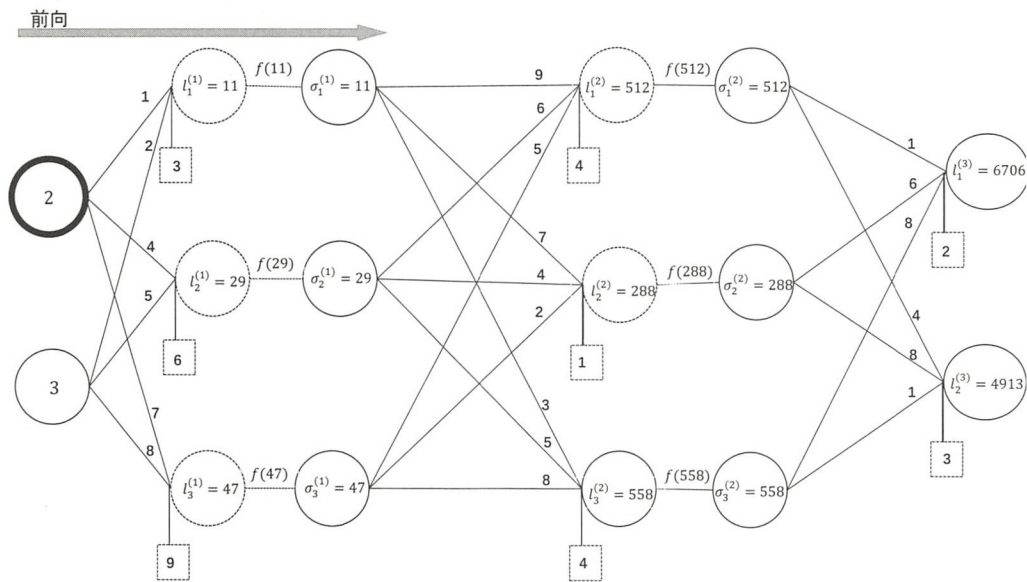
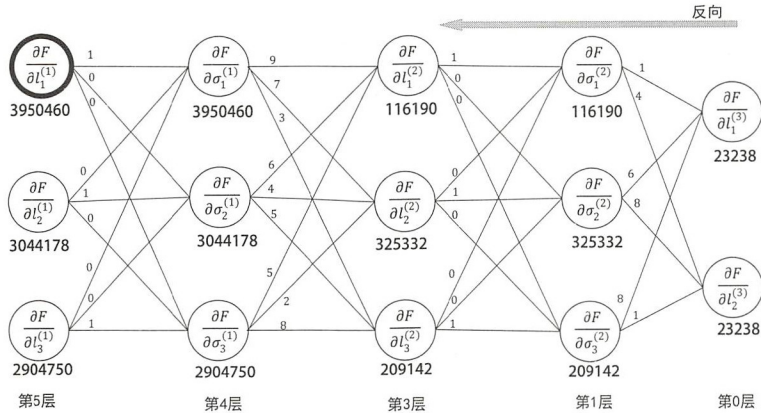


图 6-22 第 0 层的第 1 个神经元的值

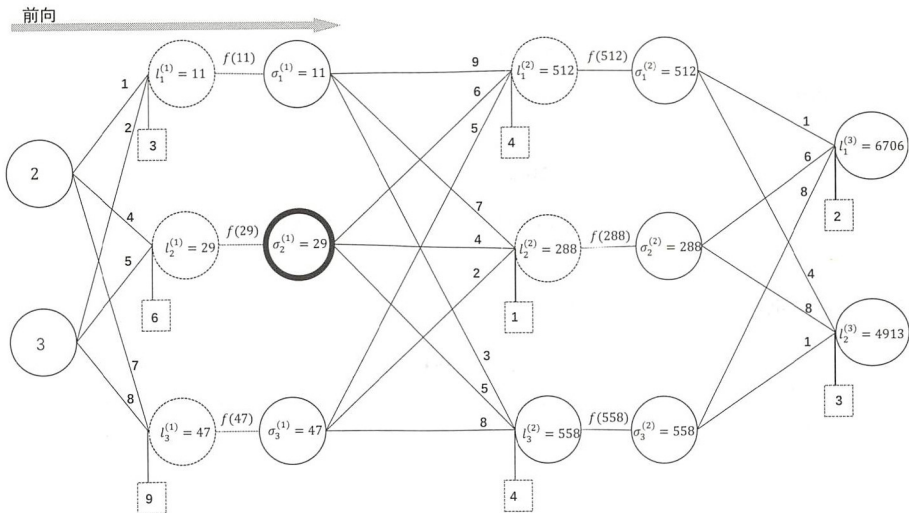
然后，根据图 6-21 找到 $\frac{\partial F}{\partial l_1^{(1)}}$ 处的值，如图 6-23 所示。

图 6-23 找到 $\frac{\partial F}{\partial l_1^{(1)}}$ 处的值

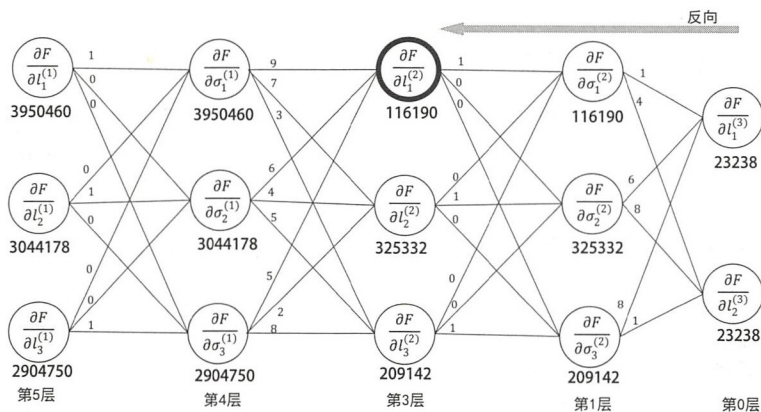
则

$$\frac{\partial F}{\partial w_{11}^{(1)} | \mathbf{wb}, w_{11}^{(1)}=1} = 2 \times 3950460 = 7900920$$

例如, 计算 F 在点 \mathbf{wb} 处关于 $w_{21}^{(2)}$ 的偏导数 $\frac{\partial F}{\partial w_{21}^{(2)} | \mathbf{wb}, w_{21}^{(2)}=6}$, 首先根据图 6-20 找到 $\sigma_2^{(1)}$ 处的值 (即第 1 层的第 2 个神经元的值), 如图 6-24 所示。

图 6-24 找到 $\sigma_2^{(1)}$ 处的值 (即第 1 层的第 2 个神经元的值)

然后, 根据图 6-21 找到 $\frac{\partial F}{\partial l_1^{(2)}}$ 处的值, 如图 6-25 所示。

图 6-25 找到 $\frac{\partial F}{\partial l_1^{(2)}}$ 处的值

则

$$\frac{\partial F}{\partial w_{21}^{(2)} | \mathbf{wb}, w_{21}^{(2)}=6} = 29 \times 116190 = 3369510$$

根据上述规律，我们可以快速地计算出函数 F 在点 \mathbf{wb} 处关于第 0 层到第 1 层的权重的偏导数：

$$\begin{aligned} \frac{\partial F}{\partial \mathbf{w}^{(1)} | \mathbf{w}^{(1)}= \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \end{bmatrix}} &= \begin{bmatrix} 2 \times 3950460 & 2 \times 3044178 & 2 \times 2904750 \\ 3 \times 3950460 & 3 \times 3044178 & 3 \times 2904750 \end{bmatrix} \\ &= \begin{bmatrix} 7900920 & 6088356 & 5809500 \\ 11851380 & 9132534 & 8714250 \end{bmatrix} \end{aligned}$$

函数 F 在点 \mathbf{wb} 处关于第 1 层到第 2 层的权重的偏导数：

$$\begin{aligned} \frac{\partial F}{\partial \mathbf{w}^{(2)} | \mathbf{w}^{(2)}= \begin{bmatrix} 9 & 7 & 3 \\ 6 & 4 & 5 \\ 5 & 2 & 8 \end{bmatrix}} &= \begin{bmatrix} 11 \times 116190 & 11 \times 325332 & 11 \times 209142 \\ 29 \times 116190 & 29 \times 325332 & 29 \times 209142 \\ 47 \times 116190 & 47 \times 325332 & 47 \times 209142 \end{bmatrix} \\ &= \begin{bmatrix} 1278090 & 3578652 & 2300562 \\ 3369510 & 9434628 & 6065118 \\ 5460930 & 15290604 & 9829674 \end{bmatrix} \end{aligned}$$

函数 F 在点 wb 处关于第 2 层到第 3 层的权重的偏导数:

$$\frac{\partial F}{\partial w^{(3)}} \bigg|_{w^{(3)} = \begin{bmatrix} 1 & 4 \\ 6 & 8 \\ 8 & 1 \end{bmatrix}} = \begin{bmatrix} 512 \times 23238 & 512 \times 23238 \\ 288 \times 23238 & 288 \times 23238 \\ 558 \times 23238 & 558 \times 23238 \end{bmatrix} = \begin{bmatrix} 11897856 & 11897856 \\ 6692544 & 6692544 \\ 12966804 & 12966804 \end{bmatrix}$$

至此, 我们已经计算出了 F 在点 wb 处的梯度, 我们可以利用函数 `gradients` 计算以上示例的梯度, 对应代码如下:

```
# -*- coding: utf-8 -*-
import tensorflow as tf
import numpy as np
#"输入层, 第0层"
x=tf.placeholder(tf.float32,shape=[None,2])
#"第0层到第1层的权重"
w1=tf.Variable(
    tf.constant(
        [
            [1,4,7],
            [2,5,8]
        ],tf.float32
    )
)
#"第1层的偏置"
b1=tf.Variable(tf.constant([3,6,9],tf.float32))
#"第1层的线性组合"
l1=tf.matmul(x,w1)+b1
#"第1层线性组合的激活"
sigma1=l1
#"第1层到第2层的权重"
w2=tf.Variable(
    tf.constant(
        [
            [9,7,3],
            [6,4,5],
            [5,2,8]
        ],tf.float32
    )
)
```

```

    )
    )
    # "第2层的偏置"
    b2=tf.Variable(tf.constant([4,1,4],tf.float32))
    # "第2层的线性组合"
    l2=tf.matmul(sigma1,w2)+b2
    # "第2层线性组合的激活"
    sigma2=l2
    # "第2层到第3层的权重"
    w3=tf.Variable(
        tf.constant(
            [
                [1,4],
                [6,8],
                [8,1]
            ],tf.float32
        )
    )
    # "第3层的偏置"
    b3=tf.Variable(tf.constant([2,3],tf.float32))
    # "第3层的线性组合，输出层"
    l3=tf.matmul(sigma2,w3)+b3
    # "构造函数F"
    F=tf.pow(tf.reduce_sum(l3),2.0)
    session=tf.Session()
    session.run(tf.global_variables_initializer())
    # "计算梯度"
    w1_gra,b1_gra,w2_gra,b2_gra,w3_gra,b3_gra=tf.gradients(F,[w1,b1,w2,b2,w3,b3])
    w1_gra_arr,b1_gra_arr,w2_gra_arr,b2_gra_arr,w3_gra_arr,b3_gra_arr=session.run(
        [w1_gra,b1_gra,w2_gra,b2_gra,w3_gra,b3_gra],feed_dict=
        {x:np.array([[2,3]],np.float32)})
    print("'关于第1层权重的梯度: '")
    print(w1_gra_arr)
    print("'关于第1层偏置的梯度: '")
    print(b1_gra_arr)
    print("'关于第2层权重的梯度: '")

```

```

print(w2_gra_arr)
print("'关于第2层偏置的梯度: '")
print(b2_gra_arr)
print("'关于第3层权重的梯度'")
print(w3_gra_arr)
print("'关于第3层偏置的梯度: '")
print(b3_gra_arr)

```

打印结果如下：

```

"关于第1层权重的梯度:"
[[ 7900920.  6088356.  5809500.]
 [11851380.  9132534.  8714250.]]
"关于第1层偏置的梯度:"
[3950460. 3044178. 2904750.]
"关于第2层权重的梯度:"
[[ 1278090.  3578652.  2300562.]
 [ 3369510.  9434628.  6065118.]
 [ 5460930. 15290604.  9829674.]]
"关于第2层偏置的梯度:"
[116190. 325332. 209142.]
"关于第3层权重的梯度"
[[11897856. 11897856.]
 [ 6692544.  6692544.]
 [12966804. 12966804.]]
"关于第3层偏置的梯度:"
[23238. 23238.]

```

从打印结果可以看出，手动计算和程序打印的结果是相等的。

以上示例中的全连接神经网络只有一个输入，如果有多个输入，可以针对每一个输入的输出生构造一个函数 F_i ，若函数 $F = \sum_i F_i$ ，则 F 针对权重和偏置的偏导数等于每一个 F_i 针对权重和偏置偏导数的和，即只要针对每一个 F_i 利用梯度反向传播算法计算权重和偏导数，然后求和即可。仔细观察就会发现针对分类问题构造的损失函数可以理解多个函数的和，所以可以利用梯度反向传播算法训练网络。

理解了全连接神经网络的梯度反向传播，接下来我们介绍训练全连接神经网络过程中常遇到的一个问题：梯度消失。

6.4.2 梯度消失

我们先通过一个简单的示例来介绍什么是梯度消失。假设有如图 6-26 所示的全连接神经网络，每一层都只有一个神经元，激活函数记为 $f(\cdot)$ 。

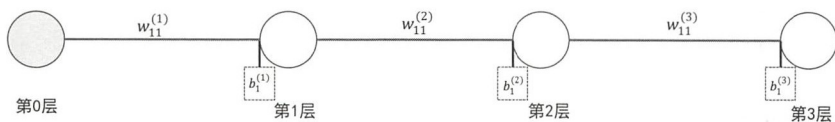


图 6-26 全连接神经网络

针对上述网络有一个输入，假设为 2 或者任何一个已知数，根据全连接神经网络的计算方法，我们可以很快计算出针对该输入在每一层每个神经元的值，如图 6-27 所示。

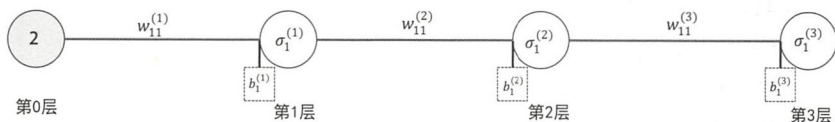


图 6-27 根据输入值计算每一层每个神经元的值

为了便于讨论，我们把图 6-27 所示的每一层的线性组合展示出来，如图 6-28 所示。

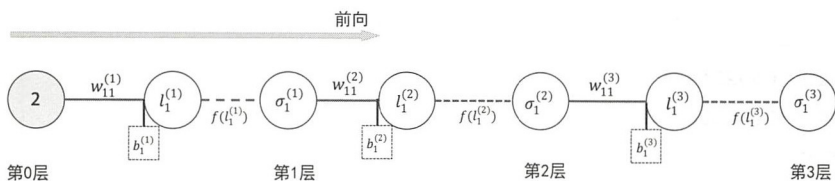


图 6-28 线性组合和激活值

根据网络的输出值 $\sigma^{(3)}$ 构造一个函数 F ，假设 $F = (\sigma^{(3)})^2$ ，该函数针对中间变量的偏导数满足梯度反向传播，如图 6-29 所示。

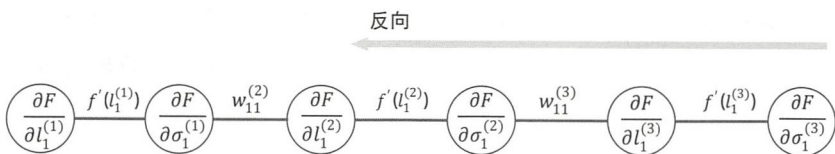


图 6-29 导数的反向传播

我们计算 F 关于第一层偏置 $b_1^{(1)}$ 的偏导数, 根据导数的链式法则和反向传播, 可知

$$\frac{\partial F}{\partial b_1^{(1)}} = \frac{\partial F}{\partial l_1^{(1)}}$$

$\frac{\partial F}{\partial l_1^{(1)}}$ 又可以根据图 6-29 所示的全连接神经网络计算出来, 即

$$\frac{\partial F}{\partial l_1^{(1)}} = \frac{\partial F}{\partial \sigma_1^{(3)}} \times f'(l_1^{(3)}) \times \frac{\partial F}{\partial l_1^{(3)}} \times w_{11}^{(3)} \times \frac{\partial F}{\partial \sigma_1^{(2)}} \times f'(l_1^{(2)}) \times \frac{\partial F}{\partial l_1^{(2)}} \times w_{11}^{(2)} \times \frac{\partial F}{\partial \sigma_1^{(1)}} \times f'(l_1^{(1)})$$

如果这里采用的激活函数是 5.4 节介绍的 sigmoid, 该激活函数在每一点的梯度的范围是 $(0, 0.25]$, 那么上式在计算 $\frac{\partial F}{\partial l_1^{(1)}}$ 的过程中, 出现了 3 次 sigmoid 函数导数的相乘, 3 个范围是 $(0, 0.25]$ 的值相乘, 就是一个很小的数。

对于上述问题, 如果网络层数很多, N 个 sigmoid 激活函数导数的相乘得到的结果更小, 那么 $\frac{\partial F}{\partial b_1^{(1)}}$ 就有可能非常小, 甚至接近 0。在进行梯度下降时, 每一次迭代, $b_1^{(1)}$ 值的变化很小, 该问题就是常称的梯度消失。显然, 越靠近输入层的权重和偏置越可能出现梯度消失问题。显然, 5.4 节介绍的 ReLU 激活函数及针对它的变形的激活函数, 可以有效地解决训练网络过程中出现的梯度消失问题, 而且该函数的计算复杂度比 sigmoid 激活函数低。ReLU 激活函数也有一个缺点, 仍以上述示例为例, 例如要计算 F 关于 $w_{11}^{(2)}$ 的偏导数, 根据图 6-28 和图 6-29, 可得 $\frac{\partial F}{\partial w_{11}^{(2)}} = \sigma_1^{(1)} \times \frac{\partial F}{\partial l_1^{(2)}}$, 假设在某一点, $l_1^{(2)} < 0$, 那么 $\text{relu}'(l_1^{(2)})$ 等于 0, 根据导数的反向传播算法, 在该点处针对 $l_1^{(2)}$ 的偏导数 $\frac{\partial F}{\partial l_1^{(2)}} = 0$, 那么 F 在该点处关于 $w_{11}^{(2)}$ 的偏导数等于 0, 所以在进行梯度下降时, 导致 $w_{11}^{(2)}$ 不更新了。解决办法很简单, 在 $l_1^{(2)} < 0$ 时, 找到一个激活函数使得在该点处的导数不等 0 (比 0 大一点) 即可, 这就是 5.4 节提出的针对 ReLU 的变形函数, 如 leaky relu 激活函数, 该函数在小于 0 的点的导数等于一个比 0 稍大的数。因此, ReLU 系列的激活函数成为神经网络中最常用的激活函数。

本章, 我们比较全面地介绍了全连接神经网络是如何处理分类问题的, 以及其对应的 TensorFlow 实现。从第 7 章开始, 作者将介绍关于卷积神经网络的内容, 从名字就可以看出该网络结构和卷积有直接关系, 所以, 作者先详细介绍什么是卷积及其对应的 TensorFlow 实现。

7

一维离散卷积

一维离散卷积的运算是一种主要基于向量的计算方式，本章我们介绍其计算原理及 TensorFlow 的具体实现。

7.1 一维离散卷积的计算原理

一维离散卷积通常有三种卷积类型：full 卷积、same 卷积和 valid 卷积，我们以图 7-1 所示的长度为 5 的一维张量 I 和长度为 4 的一维张量 K 为例，介绍这三种卷积的计算原理。

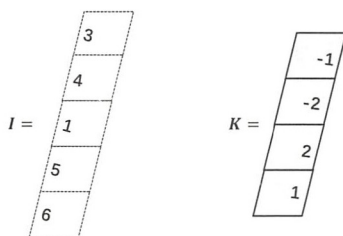
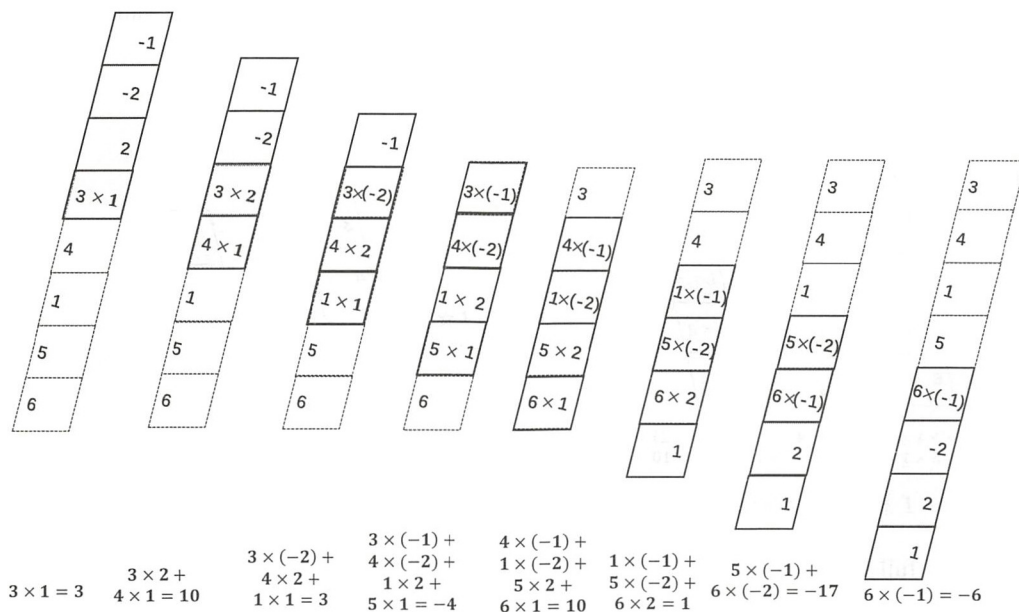


图 7-1 一维张量 I 和 K

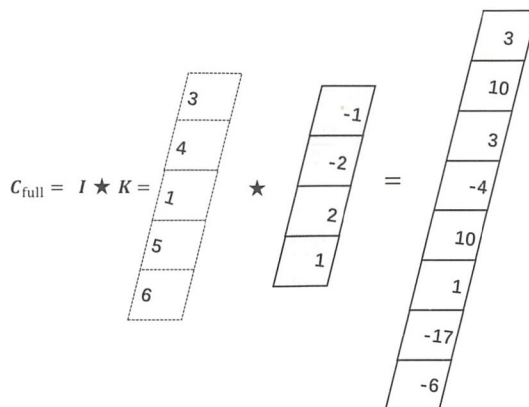
我们先介绍这两个一维张量的 full 卷积。

7.1.1 full 卷积

full 卷积的计算过程如下： \mathbf{K} 沿着 \mathbf{I} 顺序移动，每移动到一个固定位置，对应位置的值相乘，然后对其求和，过程如图 7-2 所示。

图 7-2 \mathbf{I} 和 \mathbf{K} 的 full 卷积过程

将得到的值依次存入一维张量 \mathbf{C}_{full} ，该张量就是 \mathbf{I} 和 \mathbf{K} 的 full 卷积结果，其中 \mathbf{K} 称为卷积核或者滤波器或者卷积掩码，full 卷积用符号 \star 表示，记 $\mathbf{C}_{\text{full}} = \mathbf{I} \star \mathbf{K}$ ，如图 7-3 所示。

图 7-3 \mathbf{I} 和 \mathbf{K} 的 full 卷积结果

接着我们介绍这两个一维张量的 valid 卷积。

7.1.2 valid 卷积

从 full 卷积的计算过程可知，如果 \mathbf{K} 靠近 \mathbf{I} ，就会有部分延伸到 \mathbf{I} 之外，valid 卷积只考虑 \mathbf{I} 能完全覆盖 \mathbf{K} 内的情况，即 \mathbf{K} 在 \mathbf{I} 内部移动的情况，如图 7-4 所示。

valid 卷积用符号 \star 表示，两者的 valid 卷积结果如图 7-5 所示。

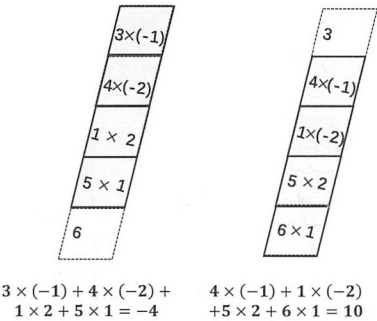


图 7-4 \mathbf{I} 和 \mathbf{K} 的 valid 卷积过程

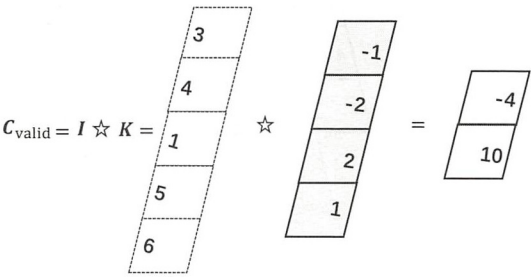


图 7-5 \mathbf{I} 和 \mathbf{K} 的 valid 卷积结果

显然，full 卷积结果的长度值比输入张量 \mathbf{I} 的长度值大，而 valid 卷积结果的长度值又比 \mathbf{I} 的长度值小。7.1.3 节将介绍的 same 卷积，其结果的长度值与 \mathbf{I} 的长度值相等。

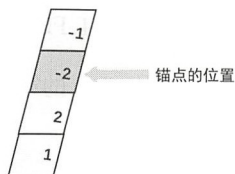
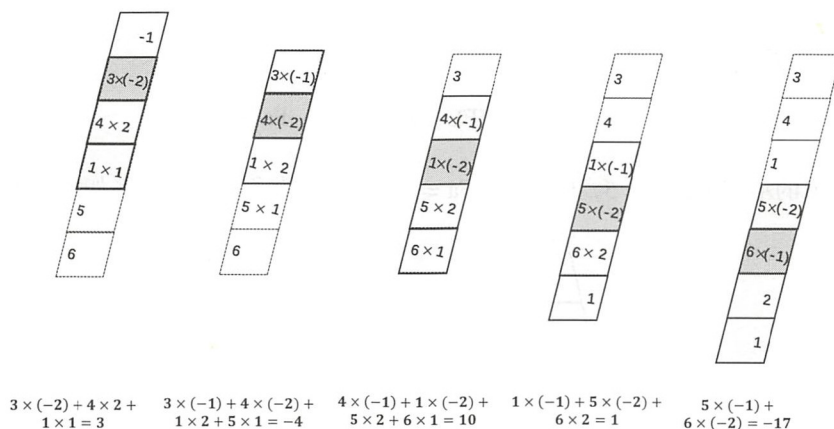
7.1.3 same 卷积

首先在卷积核 \mathbf{K} 上指定一个锚点，然后将锚点顺序移动到输入张量 \mathbf{I} 的每一个位置处，对应位置相乘然后求和。卷积核锚点的位置一般有以下规则，假设卷积核的长度为 FL ：

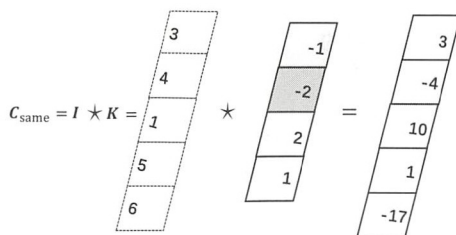
- 如果 FL 为奇数，则锚点的位置在 $\frac{\text{FL}-1}{2}$ 处
- 如果 FL 为偶数，则锚点的位置在 $\frac{\text{FL}-2}{2}$ 处

这里的位置索引是从 0 开始的。

仍用 7.1.2 节的示例，因为卷积核 \mathbf{K} 的长度为 4，4 是偶数，所以锚点的位置在 \mathbf{K} 的第 $\frac{4-2}{2} = 1$ 个位置处，如图 7-6 所示。将锚点的位置顺序移动到 \mathbf{I} 的每一个位置处，将对应位置的值相乘后求和，过程如图 7-7 所示。

图 7-6 卷积核 K 的锚点位置图 7-7 I 和 K 的 same 卷积过程

符号 \star 表示 same 卷积，两者的 same 卷积结果如图 7-8 所示。

图 7-8 I 和 K 的 same 卷积结果

从上述三类卷积的计算过程可以看出，same 卷积结果是 full 卷积结果的一部分，valid 卷积结果又是 same 卷积结果的一部分。7.1.4 节将介绍这三者的具体关系。

7.1.4 full、same、valid 卷积的关系

假设一个长度为 L 的一维张量与一个长度为 FL 的卷积核卷积，其中 Fa 代表计算 same 卷积时，锚点的位置索引，则两者的 full 卷积 C_{full} 与 same 卷积 C_{same} 的关系如下：

$$C_{\text{same}} = C_{\text{full}}[FL - Fa - 1, FL - Fa + L - 2]$$

其中 $C_{\text{full}}[FL - Fa - 1, FL - Fa + L - 2]$ 代表从 C_{full} 的第 $FL - Fa - 1$ 个值到第 $FL - Fa + L - 2$ 个值。

full 卷积 C_{full} 与 valid 卷积 C_{valid} 的关系如下：

$$C_{\text{valid}} = C_{\text{full}}[FL - 1, L - 1]$$

same 卷积 C_{same} 与 valid 卷积 C_{valid} 的关系如下：

$$C_{\text{valid}} = C_{\text{same}}[FL - Fa - 2, L - Fa - 2]$$

在 7.1 节的示例中， $L = 5$ ， $FL = 4$ ， $Fa = 1$ ，则三种卷积类型的关系如图 7-9 所示。

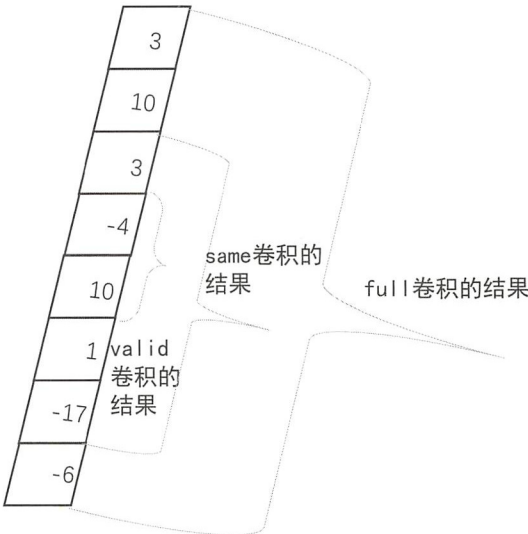


图 7-9 三种卷积类型的关系

即

$$C_{\text{same}} = C_{\text{full}}[2, 6], C_{\text{valid}} = C_{\text{full}}[3, 4], C_{\text{valid}} = C_{\text{same}}[1, 2]$$

需要注意的是，大部分书籍中对卷积运算的定义分为两步。第 1 步，将卷积核翻转 180° ；第 2 步，将翻转 180° 后的结果沿输入张量顺序移动，每移动到一个固定位置，对应位置相乘然后求和，如 Numpy 中实现的卷积函数 `convolve` 和 Scipy 中实现的卷积函数 `convolve`，函数内部都进行了上述两步运算。可见，最本质的卷积运算还是在第 2 步。TensorFlow 中实现的卷积函数

```
tf.nn.conv1d(value, filters, stride, padding, use_cudnn_on_gpu=None,
             data_format=None, name=None)
```

其内部就没有进行第 1 步操作，而是直接进行了第 2 步操作。上述示例对应的具体代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
#"输入张量I"
I=tf.constant(
    [
        [[3],[4],[1],[5],[6]]
    ],tf.float32
)
#"卷积核"
K=tf.constant(
    [
        [[-1]],[[-2]],[[2]],[[1]]
    ],tf.float32
)
#"same卷积"
I_conv1d_K=tf.nn.conv1d(I,K,1,'SAME')
#"创建会话"
session=tf.Session()
#"打印结果"
print(session.run(I_conv1d_K))
```

打印结果如下：

```
[[[ 3.]
   [-4.]
   [ 10.]
   [ 1.]
  [-17.]]]
```

函数 `tf.nn.conv1d` 只实现了 `same` 卷积和 `valid` 卷积，它就是为更方便地搭建卷积神经网络而设计的，其实它可以实现更丰富的功能，后续章节会介绍该函数具体的使用方法。利用 Numpy 或者 Scipy 中的卷积函数 `convolve` 实现上述示例的 `full` 卷积代码如下：

```
import numpy as np
```

```

from scipy import signal
I=np.array([3,4,1,5,6],np.float32)
K=np.array([-1,-2,2,1],np.float32)
#"卷积核K翻转180°"
K_reverse=np.flip(K,0)
#r=np.convolve(I,K_reverse,mode='full')
r=signal.convolve(I,K_reverse,mode='full')
print(r)

```

打印结果如下：

```
[ 3.  10.   3.  -4.  10.   1. -17.  -6.]
```

注意：在上述代码中，为了得到和 TensorFlow 实现相等的结果，我们要先将卷积核翻转 180° 。如果卷积核的长度是偶数，函数 `convolve` 和 `tf.nn.conv1d` 在实现 `same` 卷积时，其结果会略有不同，但也只是在边界处（两端）的值有所不同，这是因为这两个函数对卷积核锚点的位置定义不同，本质上就是从 `full` 卷积结果中取的区域不一样。以上是利用卷积的定义进行计算，7.2 节将介绍如何利用一维的傅里叶变换计算一维卷积，该过程称为**卷积定理**。

7.2 一维卷积定理

7.2.1 一维离散傅里叶变换

我们先介绍一维傅里叶变换的数学演算过程，假设已知长度为 M 的一维离散数列：

$$f(x), x = 0, 1, 2, 3, \dots, M-1$$

问题：是否存在一维离散数列

$$F(u), x = 0, 1, 2, 3, \dots, M-1$$

满足

$$f(x) = \frac{1}{M} \sum_{u=0}^{M-1} F(u) e^{\frac{2\pi}{M} u x i}, x = 0, 1, 2, 3, \dots, M-1$$

答案是肯定的，下面计算数列 $\{F(u)\}$ ，假设计算数列 $\{F(u)\}$ 的第 u_1 个数，将上式的等式两边同乘以 $e^{-\frac{2\pi}{M} u_1 x i}$ ，即

$$\begin{aligned}
f(x)e^{-\frac{2\pi}{M}u_1xi} &= \frac{1}{M} \sum_{u=0}^{M-1} F(u)e^{\frac{2\pi}{M}(u-u_1)xi} \\
&= \frac{1}{M} F(0)e^{\frac{2\pi}{M}(0-u_1)xi} + \frac{1}{M} F(1)e^{\frac{2\pi}{M}(1-u_1)xi} + \cdots + \frac{1}{M} F(u_1)e^{\frac{2\pi}{M}(u_1-u_1)xi} + \\
&\quad \cdots + \frac{1}{M} F(M-1)e^{\frac{2\pi}{M}((M-1)-u_1)xi}
\end{aligned}$$

然后, 针对上式, 等式两边针对 x 求和, 即

$$\begin{aligned}
\sum_{x=0}^{M-1} f(x)e^{-\frac{2\pi}{M}u_1xi} &= \frac{1}{M} \sum_{x=0}^{M-1} F(0)e^{\frac{2\pi}{M}(0-u_1)xi} + \frac{1}{M} \sum_{x=0}^{M-1} F(1)e^{\frac{2\pi}{M}(1-u_1)xi} + \cdots + \\
&\quad \frac{1}{M} \sum_{x=0}^{M-1} F(u_1)e^{\frac{2\pi}{M}(u_1-u_1)xi} + \cdots + \frac{1}{M} \sum_{x=0}^{M-1} F(M-1)e^{\frac{2\pi}{M}((M-1)-u_1)xi}
\end{aligned}$$

当 $u \neq u_1$ 时, $\sum_{x=0}^{M-1} F(u)e^{\frac{2\pi}{M}(u-u_1)xi} = 0$, 所以上式可简化为

$$\sum_{x=0}^{M-1} f(x)e^{-\frac{2\pi}{M}u_1xi} = F(u_1)$$

即

$$F(u_1) = \sum_{x=0}^{M-1} f(x)e^{-\frac{2\pi}{M}u_1xi}, \quad u_1 = 0, 1, 2, 3, \dots, M-1$$

因为上式对任意的 u_1 , $u_1 = 0, 1, 2, 3, \dots, M-1$ 均成立, 所以我们找到了数列 $\{F(u)\}$, 即

$$F(u) = \sum_{x=0}^{M-1} f(x)e^{-\frac{2\pi}{M}uxi}, \quad u = 0, 1, 2, 3, \dots, M-1$$

那么 F 称为 f 的傅里叶变换, 而 f 称为 F 的傅里叶逆变换, 常表示为 $f \Leftrightarrow F$ 。

对于一维离散的傅里叶变换的计算过程, 通过以下简单的示例理解, 假设已知长度为 3 的离散数列 $\{f(x)\} = \{4, 5, 6\}$, 计算该数列的傅里叶变换过程如下:

$$\begin{aligned}
F(0) &= 4e^{-\frac{2\pi}{3} \times 0 \times 0i} + 5e^{-\frac{2\pi}{3} \times 0 \times 1i} + 6e^{-\frac{2\pi}{3} \times 0 \times 2i} = 15 \\
F(1) &= 4e^{-\frac{2\pi}{3} \times 1 \times 0i} + 5e^{-\frac{2\pi}{3} \times 1 \times 1i} + 6e^{-\frac{2\pi}{3} \times 1 \times 2i} \\
&= 4 + 5\left(\cos\left(-\frac{2\pi}{3}\right) + \sin\left(-\frac{2\pi}{3}\right)i\right) + 6\left(\cos\left(-\frac{4\pi}{3}\right) + \sin\left(-\frac{4\pi}{3}\right)i\right) \\
&= 4 - 5 \times \frac{1}{2} - 6 \times \frac{1}{2} + \left(-5 \times \frac{\sqrt{3}}{2} + 6 \times \frac{\sqrt{3}}{2}\right)i = -1.5 + \frac{\sqrt{3}}{2}i \\
F(2) &= 4e^{-\frac{2\pi}{3} \times 2 \times 0i} + 5e^{-\frac{2\pi}{3} \times 2 \times 1i} + 6e^{-\frac{2\pi}{3} \times 2 \times 2i}
\end{aligned}$$

$$\begin{aligned}
&= 4 + 5 \left(\cos \left(-\frac{4\pi}{3} \right) + \sin \left(-\frac{4\pi}{3} \right) i \right) + 6 \left(\cos \left(-\frac{8\pi}{3} \right) + \sin \left(-\frac{8\pi}{3} \right) i \right) \\
&= 4 - 5 \times \frac{1}{2} - 6 \times \frac{1}{2} + \left(5 \times \frac{\sqrt{3}}{2} - 6 \times \frac{\sqrt{3}}{2} \right) i = -1.5 - \frac{\sqrt{3}}{2} i
\end{aligned}$$

结果如图 7-10 所示。

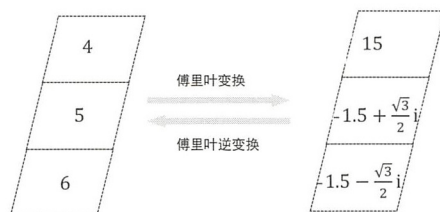


图 7-10 傅里叶变换与逆变换

TensorFlow 通过函数 `fft` 和 `ifft` 分别实现一维离散的傅里叶变换及逆变换，以上示例的代码如下：

```

# -*- coding: utf-8 -*-
import tensorflow as tf
#"输入长度为3的一维张量"
f=tf.constant([4,5,6],tf.complex64)
#"创建会话"
session=tf.Session()
#"一维傅里叶变换"
F=tf.fft(f)
#"打印傅里叶变换F的值"
print("傅里叶变换F的值:")
print(session.run(F))
#"计算F的傅里叶逆变换(显然与输入的f是相等的)"
F_ifft=tf.ifft(F)
#"打印F的傅里叶逆变换的值"
print("打印F的傅里叶逆变换的值:")
print(session.run(F_ifft))

```

打印结果如下：

```

"傅里叶变换F的值:"
[ 15.00000191 -2.98023224e-07j -1.49999952 +8.66027594e-01j
 -1.50000191 -8.66023958e-01j]

```

"打印F的傅里叶逆变换的值:"

```
[ 4.000000000 +1.21196115e-06j  5.000000095 -1.58945724e-07j
 6.00000191 -9.53674316e-07j]
```

理解了一维离散傅里叶（逆）变换的计算过程，接下来我们介绍如何利用一维离散的傅里叶变换计算一维离散卷积。

7.2.2 卷积定理

假设有长度为 L 的一维张量 I ， $I(l)$ 代表 I 的第 l 个数，其中 $0 \leq l < L$ ，有长度为 FL 的一维卷积核 K ，那么 I 与 K 的 full 卷积结果的尺寸为 $L + FL - 1$ 。

首先，在 I 的末尾补零，将 I 的尺寸扩充到与 full 卷积的尺寸相同，即

$$I_{\text{padded}}(l) = \begin{cases} I(l), & 0 \leq l < L \\ 0, & L \leq l < (L + FL - 1) - 1 \end{cases}$$

然后，将卷积核 K 翻转 180° 得到 $K_{\text{rotate180}}$ ，在末尾进行补零操作，且将 $K_{\text{rotate180}}$ 的尺寸扩充到和 full 卷积相同，即

$$K_{\text{rotate180_padded}}(l) = \begin{cases} K_{\text{rotate180}}(l), & 0 \leq l < FL \\ 0, & FL \leq l < (L + FL - 1) - 1 \end{cases}$$

假设 fft_Ip 和 fft_Krp 分别是 I_{padded} 和 $K_{\text{rotate180_padded}}$ 的傅里叶变换，那么 $I \star K$ 的傅里叶变换等于 $\text{fft_Ip} * \text{fft_Krp}$ ，即

$$I \star k \Leftrightarrow \text{fft_Ip} * \text{fft_Krp}$$

其中 $*$ 代表对应元素相乘，即对应位置的两个复数相乘，该关系通常称为卷积定理。

我们通过 TensorFlow 提供的函数验证上述过程，从卷积定理中可以看出分别有对张量的补零操作（或称为边界扩充）和翻转操作。这两种操作在 TensorFlow 中有对应的函数实现，我们先介绍实现边界扩充的函数：

```
pad(tensor, paddings, mode='CONSTANT', name=None, constant_values=0)
```

以长度为 2 的一维张量为例，上侧补 1 个 0，下侧补 2 个 0，结果如图 7-11 所示。

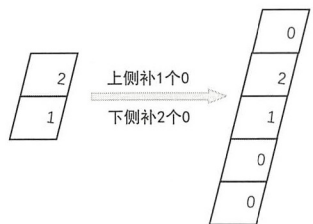


图 7-11 一维张量的边界扩充

对应的实现代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
#"一维张量"
x=tf.constant([2,1],tf.float32)
#"常数边界扩充，上侧补1个0，下侧补2个0"
r=tf.pad(x,[[1,2]],mode='CONSTANT')
#"创建会话"
session=tf.Session()
#"打印边界扩充后的结果"
print(session.run(r))
```

打印结果如下：

[0. 2. 1. 0. 0.]

当使用常数进行扩充时，也可以选择其他常数，通过参数 `constant_values` 进行设置，默认缺省值为 0。例如，在 2 行 3 列的二维张量上侧补 1 行，下侧补 2 行，右侧补 1 列，其中扩充的值为 10，结果如图 7-12 所示。

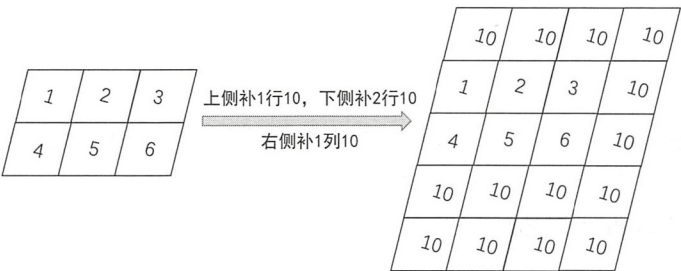


图 7-12 二维张量的边界扩充

对应的实现代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
#"二维张量"
x=tf.constant([[1,2,3],[4,5,6]],tf.float32)
#"常数边界扩充，上侧补1行10，下侧补2行10，右侧补1列10"
r=tf.pad(x,[[1,2],[0,1]],mode='CONSTANT',constant_values=10)
#"创建会话"
session=tf.Session()
#"打印边界扩充后的结果"
print(session.run(r))
```

打印结果如下：

```
[[10. 10. 10. 10.]
 [ 1.  2.  3. 10.]
 [ 4.  5.  6. 10.]
 [10. 10. 10. 10.]
 [10. 10. 10. 10.]]
```

除了常数边界扩充，还有其他扩充方式，可以通过参数 `mode` 设置，当 `mode='SYMMETRIC'` 时，代表镜像方式的边界扩充；当 `mode='REFLECT'` 时，代表反射方式的边界扩充，可以修改以上程序观察打印结果。

TensorFlow 通过函数 `reverse(tensor, axis, name=None)` 实现张量的翻转，如对以下 2 行 3 列的二维张量的每一列翻转（沿“0”方向），则称为**水平镜像**，如图 7-13 所示。同理，如果对每一行翻转（沿“1”方向），则称为**垂直镜像**，如图 7-14 所示。

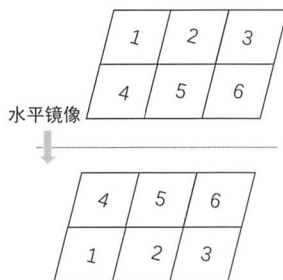


图 7-13 水平镜像

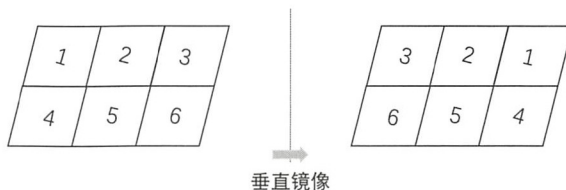


图 7-14 垂直镜像

如果先对该二维张量进行水平镜像，再进行垂直镜像，或者先进行垂直镜像，再进行水平镜像，即对其逆时针翻转 180° ，则结果如图 7-15 所示。

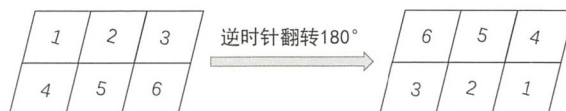


图 7-15 逆时针翻转 180°

上述过程对应的实现代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
#"二维张量"
t=tf.constant([[1,2,3],[4,5,6]],tf.float32)
#"水平镜像"
rh=tf.reverse(t,axis=[0])
#"垂直镜像"
rv=tf.reverse(t,axis=[1])
#"逆时针翻转180°: 先水平镜像再垂直镜像(或者先垂直镜像再水平镜像)"
r=tf.reverse(t,axis=[0,1])
#"创建会话"
session=tf.Session()
#"打印结果"
print('水平镜像的结果')
print(session.run(rh))
print('垂直镜像的结果')
print(session.run(rv))
print('逆时针翻转180°的结果')
print(session.run(r))
```

打印结果如下：

"水平镜像的结果"

```
[[4. 5. 6.]
 [1. 2. 3.]]
```

"垂直镜像的结果"

```
[[3. 2. 1.]
 [6. 5. 4.]]
```

"逆时针翻转180°的结果"

```
[[6. 5. 4.]
 [3. 2. 1.]]
```

掌握了张量边界扩充和翻转的对应函数后，我们利用卷积定理计算 7.1 节中 \mathbf{x} 和 \mathbf{K} 的 full 卷积，具体代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
#"长度为5的输入张量"
I=tf.constant([3,4,1,5,6],tf.complex64)
#"长度为4的卷积核"
K=tf.constant([1,2,-2,-1],tf.complex64)
#"补0操作"
I_padded=tf.pad(I,[[0,3]])
#"将卷积核翻转180°"
K_rotate180=tf.reverse(K,axis=[0])
#"翻转后进行补0操作"
K_roate180_padded=tf.pad(K_rotate180,[[0,4]])
#"傅里叶变换"
I_padded_fft=tf.fft(I_padded)
#"傅里叶变换"
K_roate180_padded_fft=tf.fft(K_roate180_padded)
#"将以上两个傅里叶变换点乘操作"
Ik_fft=tf.multiply(I_padded_fft,K_roate180_padded_fft)
#"傅里叶逆变换"
Ik=tf.ifft(Ik_fft)
#"因为输入的张量和卷积核都是实数，对以上傅里叶逆变换进行取实部的操作"
Ik_real=tf.real(Ik)
session=tf.Session()
#"打印结果"
print(session.run(Ik_real))
```

打印结果如下：

```
[ 2.99999976  10.   3.  -4.  10.   1.  -17.  -6.]
```

显然，以上结果为 \mathbf{I} 和 \mathbf{K} 的 full 卷积结果，same 卷积和 valid 卷积的结果可以根据 7.1.4 节中介绍的与 full 卷积结果的关系得到。当卷积核的维数较大时，利用卷积定理可以有效地加速计算卷积。理解了一维卷积的定义及其卷积定理后，我们着重介绍 TensorFlow 实现的卷积函数 `tf.nn.conv1d`。

7.3 具备深度的一维离散卷积

7.3.1 具备深度的张量与卷积核的卷积

具备深度的张量可以通过图 7-16 所示的张量 \mathbf{x} 和 \mathbf{K} 理解。

张量 \mathbf{x} 可以理解为是一个长度为 3、深度为 3 的张量， \mathbf{K} 可以理解为是一个长度为 2、深度为 3 的张量，两者 same 卷积的过程就是锚点顺序移动到输入张量的每一个位置处，然后对应位置相乘、求和，过程如图 7-17 所示。

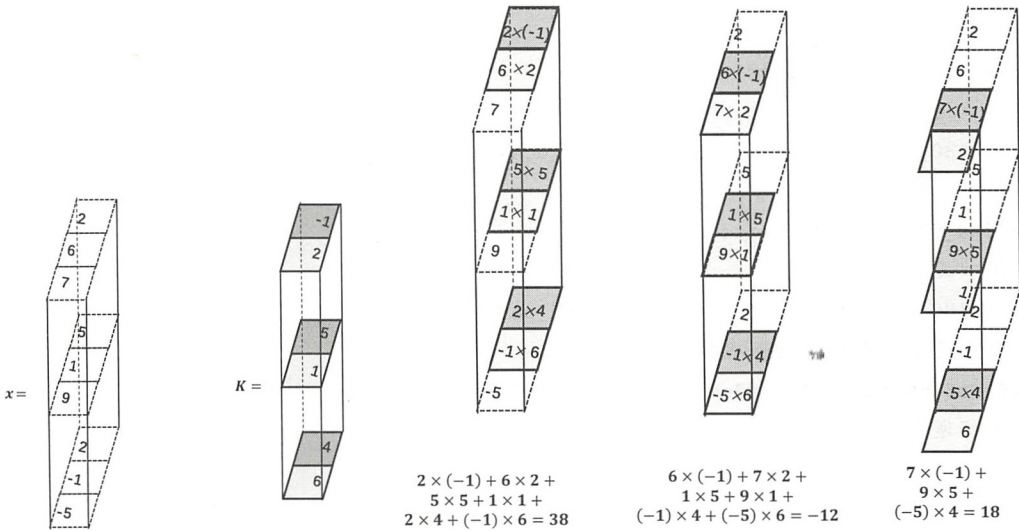


图 7-16 具备深度的张量

图 7-17 具备深度的张量的卷积过程

注意：输入张量的深度和卷积核的深度是相等的，两者 same 卷积的结果如图 7-18 所示。

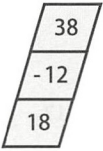


图 7-18 具备深度的张量的卷积结果

显然，具备深度的张量和卷积核的卷积过程可以理解为分别在每一层深度上进行卷积，然后在深度方向上求和，利用函数 `tf.nn.conv1d` 实现该过程的具体代码如下：

```
# -*- coding: utf-8 -*-
```

```

import tensorflow as tf
#"1个长度为3、深度为3的张量"
x=tf.constant(
    [
        [[2,5,2],[6,1,-1],[7,9,-5]]
    ],tf.float32
)
#"1个长度为2、深度为3的卷积核"
k=tf.constant(
    [
        [[-1],[5],[4]],[[2],[1],[6]]
    ],tf.float32
)
#"一维same卷积"
v_conv1d_k=tf.nn.conv1d(x,k,1,'SAME')
#"创建会话"
session=tf.Session()
#"打印结果"
print(session.run(v_conv1d_k))

```

打印结果如下：

```

[
[[ 38.],[-12.],[ 18.]]
]

```

注意：二维张量与具备深度的张量在 Tensorflow 中的表示方法一样。

以上示例是一个张量和一个卷积核进行卷积。注意，它们的深度相等才可以卷积。接下来介绍同一个张量与多个卷积核的卷积。

7.3.2 具备深度的张量分别与多个卷积核的卷积

同一个张量与多个卷积核的卷积本质上是该张量分别与每一个卷积核卷积，然后将每一个卷积结果在深度方向上连接在一起。我们以长度为3、深度为3的输入张量与2个长度为2、深度为3的卷积核卷积为例讲解，如图7-19所示。

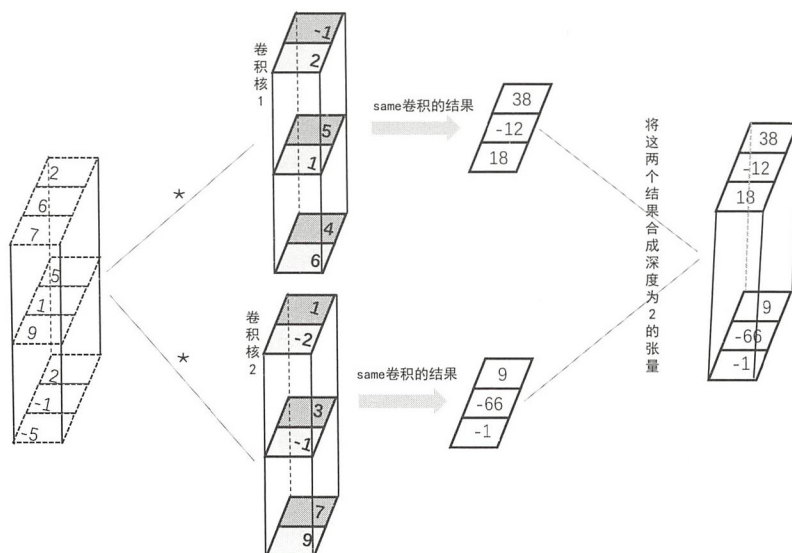


图 7-19 具备深度的张量分别与多个卷积核卷积

利用函数 `tf.nn.conv1d` 实现上述示例的具体代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
#"1个长度为3、深度为3的张量"
x=tf.constant(
    [
        [[2,5,2],[6,1,-1],[7,9,-5]]
    ],tf.float32
)
#"2个长度为2、深度为3的卷积核"
k=tf.constant(
    [
        [[-1,1],[5,3],[4,7]], [[2,-2],[1,-1],[6,9]]
    ],tf.float32
)
#"一维same卷积"
v_conv1d_k=tf.nn.conv1d(x,k,1,'SAME')
#"创建会话"
session=tf.Session()
#"打印结果"
```

```
print(session.run(v_conv1d_k))
```

打印结果如下:

```
[
[[ 38., 9.], [-12., -66.], [ 18., -1.]]
]
```

以上示例最终输出的计算结果的长度为 3、深度为 2，从计算过程可以看出，1 个深度为 C 的张量与 M 个深度为 C 的卷积核的卷积结果的深度为 M ，即最后输出结果的深度与卷积核的个数相等。

7.3.3 多个具备深度的张量分别与多个卷积核的卷积

在 7.3.2 节的示例中，我们实现了 1 个长度为 3、深度为 3 的张量与 2 个深度为 3 的卷积核的卷积，其实用函数 `conv1d` 可以方便地实现多个张量分别与多个卷积核的卷积。图 7-20 所示为 3 个长度为 3、深度为 3 的张量与 2 个长度为 2、深度为 3 的卷积核的卷积。

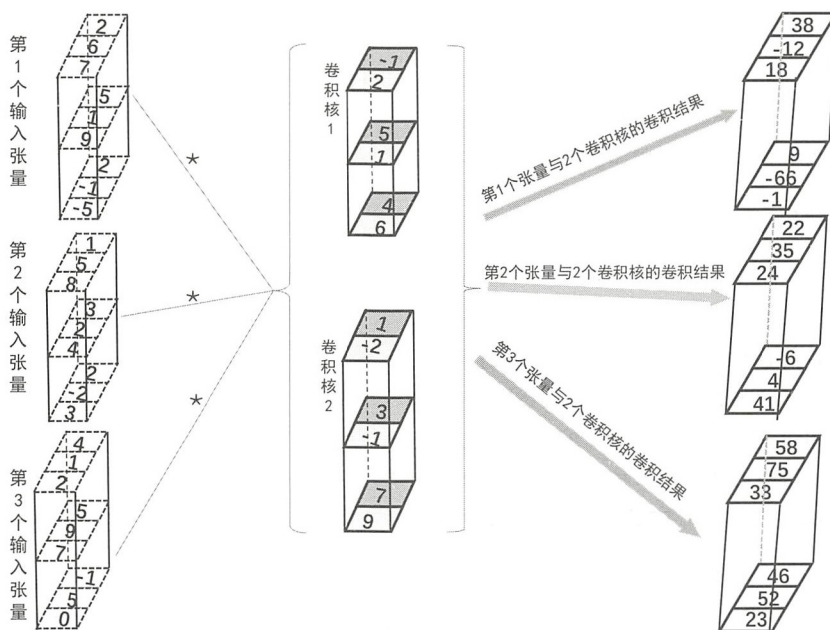


图 7-20 多个具备深度的张量分别与多个卷积核卷积

对应的具体代码实现如下:

```

# -*- coding: utf-8 -*-
import tensorflow as tf
#"3个长度为3、深度为3的张量"
x=tf.constant(
    [
        [2,5,2],[6,1,-1],[7,9,-5]], #"第1个长度为3、深度为3的张量"
        [1,3,2],[5,2,-2],[8,4,3]], #"第2个长度为3、深度为3的张量"
        [4,5,-1],[1,9,5],[2,7,0]]   #"第3个长度为3、深度为3的张量"
    ],tf.float32
)
#"2个长度为2、深度为3的卷积核"
k=tf.constant(
    [
        [-1,1],[5,3],[4,7]], [[2,-2],[1,-1],[6,9]]
    ],tf.float32
)
#"一维same卷积"
v_conv1d_k=tf.nn.conv1d(x,k,1,'SAME')
#"创建会话"
session=tf.Session()
#"打印结果"
print(session.run(v_conv1d_k))

```

打印结果如下：

```

[
#"第1个一维张量与2个卷积核的卷积结果"
[[ 38.   9.],[-12. -66.],[ 18.  -1.]],
#"第2个一维张量与2个卷积核的卷积结果"
[[ 22.  -6.],[ 35.   4.],[ 24.  41.]],
#"第3个一维张量与2个卷积核的卷积结果"
[[ 58.  46.],[ 75.  52.],[ 33.  23.]]
]

```

通过 7.3.1 ~ 7.3.3 节的示例可以看出，函数 `tf.nn.conv1d` 可以实现任意多个输入张量分别与任意多个卷积核的卷积，输入张量的深度和卷积核的深度是相等的。理解了一维张量的卷积，第 8 章将介绍比较复杂的二维张量的卷积运算。

8

二维离散卷积

8.1 二维离散卷积的计算原理

二维离散卷积的计算原理同一维离散卷积的计算原理类似，也有三种卷积类型：full 卷积、same 卷积和 valid 卷积。这三种类型的卷积计算原理，可以通过图 8-1 所示的 3 行 3 列的二维张量 \mathbf{x} 和 2 行 2 列的二维张量 \mathbf{K} 理解。

$$\mathbf{x} = \begin{bmatrix} 2 & 3 & 8 \\ 6 & 1 & 5 \\ 7 & 2 & -1 \end{bmatrix} \quad \mathbf{K} = \begin{bmatrix} 4 & 1 \\ 2 & 3 \end{bmatrix}$$

图 8-1 full 卷积过程

我们分别计算这两个二维张量的 full 卷积、same 卷积和 valid 卷积的结果。

8.1.1 full 卷积

full 卷积的计算过程如下： \mathbf{K} 沿着 \mathbf{x} 按照先行后列的顺序移动，每移动到一个固定位置，对应位置的值相乘，然后求和，过程如图 8-2 所示。

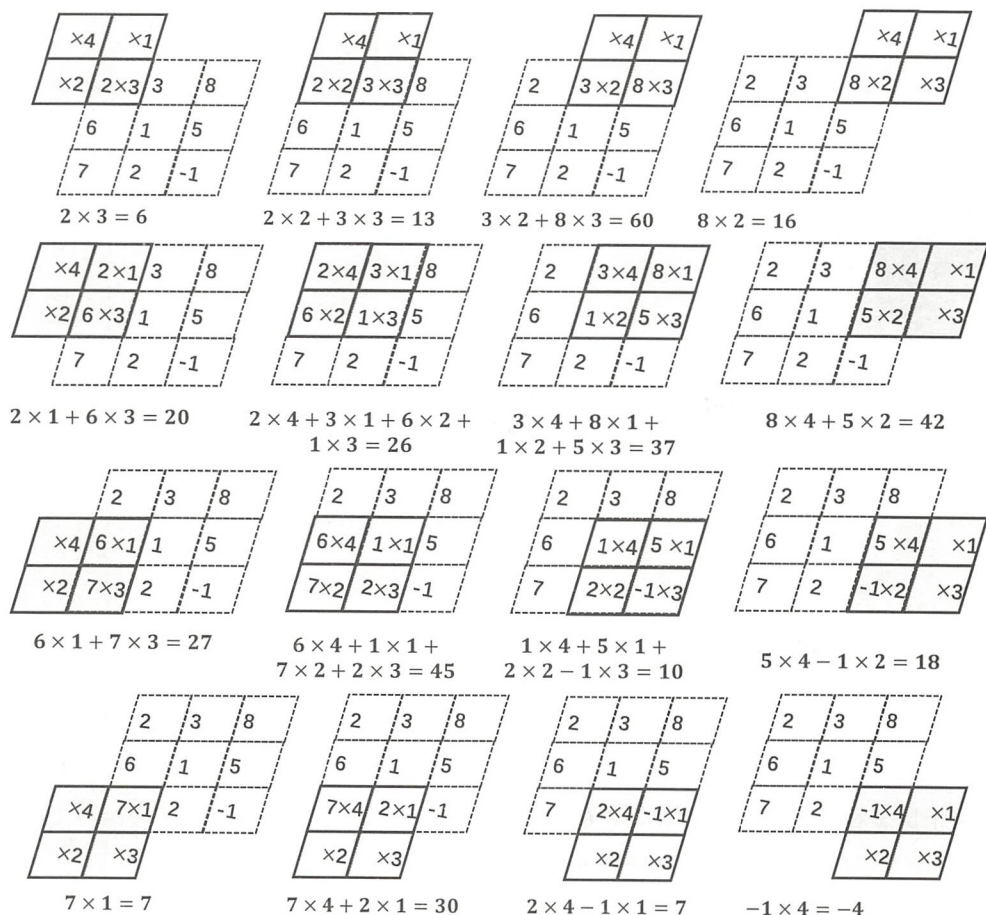


图 8-2 full 卷积的过程

其中 K 称为卷积核或者卷积掩码或者滤波器，将得到的值依次存入二维张量 C_{full} 中，该张量就是 x 和 K 的“full 卷积”结果，用符号 \star 表示，记作 $C_{\text{full}} = x \star K$ ，结果如图 8-3 所示。

$$C_{\text{full}} = x \star K = \begin{bmatrix} 2 & 3 & 8 \\ 6 & 1 & 5 \\ 7 & 2 & -1 \end{bmatrix} \star \begin{bmatrix} 4 & 1 \\ 2 & 3 \end{bmatrix} = \begin{bmatrix} 6 & 13 & 60 & 16 \\ 20 & 26 & 37 & 42 \\ 27 & 45 & 10 & 18 \\ 7 & 30 & 7 & -4 \end{bmatrix}$$

图 8-3 full 卷积的结果

注意：同一维卷积类似，对二维卷积的定义一般分为两步，首先将卷积核翻转 180° ，然后计算对应位置相乘的和，如常用的 Numpy、MATLAB 中实现的卷积函数都是先将输入的卷积核翻转 180° ，TensorFlow 中实现二维卷积的函数为

```
tf.nn.conv2d(input, filter, strides, padding, use_cudnn_on_gpu=True,
              data_format="NHWC", dilations=[1, 1, 1, 1], name=None)
```

该函数内部没有对卷积核翻转，以下讨论的关于卷积的运算及其性质都是没有对卷积核翻转的。

8.1.2 same 卷积

\mathbf{x} 与 \mathbf{K} 进行 same 卷积，首先为 \mathbf{K} 指定一个锚点，然后将锚点先后列地移动到输入张量 \mathbf{x} 的每一个位置处，对应位置相乘然后求和。卷积核 \mathbf{K} 的高等于 FH，宽等于 FW，其锚点的位置一般用以下规则定义。

- 如果 FH 为奇数，FW 为奇数，锚点的位置是 $\left(\frac{FH-1}{2}, \frac{FW-1}{2}\right)$
- 如果 FH 为奇数，FW 为偶数，锚点的位置是 $\left(\frac{FH-1}{2}, \frac{FW-2}{2}\right)$
- 如果 FH 为偶数，FW 为奇数，锚点的位置是 $\left(\frac{FH-2}{2}, \frac{FW-1}{2}\right)$
- 如果 FH 为偶数，FW 为偶数，锚点的位置是 $\left(\frac{FH-2}{2}, \frac{FW-2}{2}\right)$

这里的位置索引是从 0 开始的。

以 8.1 节的示例为例， \mathbf{K} 的高为 2、宽为 2，所以锚点的位置在 \mathbf{K} 的 (0,0) 处，如图 8-4 所示。

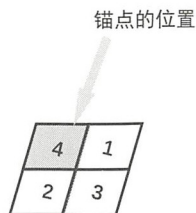


图 8-4 锚点的位置

两者的 same 卷积过程如图 8-5 所示。

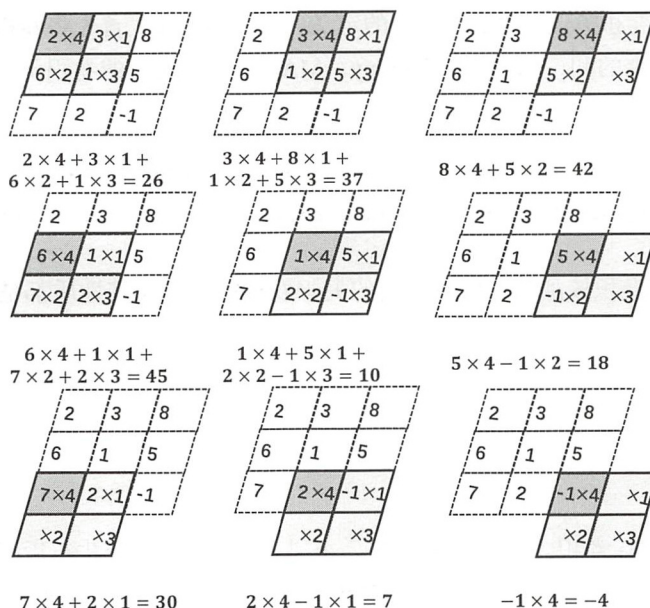


图 8-5 same 卷积的过程

将得到的值依次存入二维张量 \mathbf{C}_{same} ，该张量就是 \mathbf{x} 和 \mathbf{K} same 卷积的结果，用符号 \star 表示，记作 $\mathbf{C}_{\text{same}} = \mathbf{x} \star \mathbf{K}$ ，如图 8-6 所示。

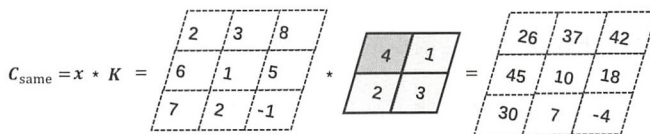


图 8-6 same 卷积的结果

利用函数 `tf.nn.conv2d` 实现上述示例的 same 卷积，具体代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
#"输入张量"
X=tf.constant(
    [
        [
            [[2],[3],[8]],
            [[6],[1],[5]],
            [[7],[2],[-1]]
        ]
    ]
)
```

```

        ],tf.float32
    )
    #"卷积核"
    K=tf.constant(
        [
            [[4]],[[1]]],
            [[2]],[[3]]]
        ],tf.float32
    )
    #"same 卷积"
    conv=tf.nn.conv2d(X,K,(1,1,1,1),'SAME')
    session=tf.Session()
    #"打印结果"
    print(session.run(conv))

```

打印结果如下：

```

[[[ 26.],[ 37.],[ 42.],
  [ 45.],[ 10.],[ 18.],
  [ 30.],[ 7.],[ -4.]]]

```

函数 `tf.nn.conv2d` 还有更多丰富的应用场景，后面会慢慢展开。

8.1.3 valid 卷积

从以上 full 卷积和 same 卷积的计算过程可知，如果卷积核 \mathbf{K} 靠近 \mathbf{x} 的边界，那么 \mathbf{K} 就会有部分延伸到 \mathbf{x} 外，导致访问到未定义的值；如果忽略边界，只考虑 \mathbf{x} 能完全覆盖 \mathbf{K} 值的情况（即 \mathbf{K} 在 \mathbf{x} 内移动），则该过程称为 valid 卷积。

仍以 8.1 节示例中的 \mathbf{x} 和 \mathbf{K} 为例，两者 valid 卷积的过程如图 8-7 所示。

将得到的值依次存入二维张量 $\mathbf{C}_{\text{valid}}$ ，该张量就是 \mathbf{x} 和 \mathbf{K} 的 valid 卷积的结果，用符号 \star 表示，记 $\mathbf{C}_{\text{valid}} = \mathbf{x} \star \mathbf{K}$ ，其结果如图 8-8 所示。

只需要将 8.1.2 节代码中的函数 `tf.nn.conv2d`，参数 `padding='SAME'` 设置为 `'VALID'`，即 `conv=`

```
tf.nn.conv2d(X,K,(1,1,1,1),'VALID')
```

打印结果如下：

```
[  
[[[ 26.],[ 37.]],  
[[ 45.],[ 10.]]]  
]
```

至此，我们了解了二维 full、same 和 valid 卷积，接下来介绍这三者的关系。

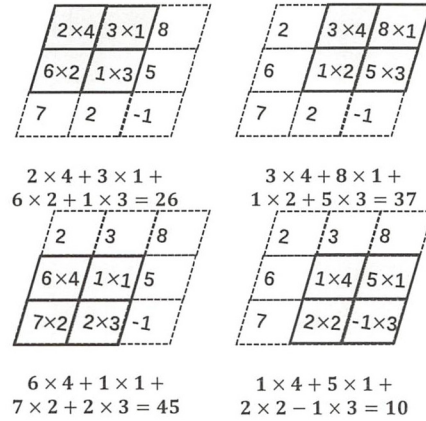


图 8-7 valid 卷积的过程

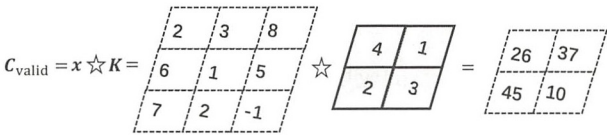


图 8-8 valid 卷积的结果

8.1.4 full、same、valid 卷积的关系

假设有 H 行 W 列的二维张量 x 与 FH 行 FW 列的二维张量 K 卷积，两者 full 卷积的结果记为 C_{full} ，same 卷积的结果记为 C_{same} ，valid 卷积的结果记为 C_{valid} 。

1. full 卷积与 valid 卷积的关系

C_{valid} 和 C_{full} 的关系为

$$C_{\text{valid}} = C_{\text{full}}[FH - 1 : H - 1, FW - 1 : W - 1]$$

即 C_{full} 的第 $FH - 1$ 行至 $H - 1$ 行、第 $FW - 1$ 列至 $W - 1$ 列的值为 C_{valid} 。注意：这里的位置索引是从 0 开始计数的。

以 8.1 节示例中的 \mathbf{x} 和 \mathbf{K} 为例, 根据图 8-3 和图 8-8 所示的两者的 full 卷积和 valid 卷积可知 $\mathbf{C}_{\text{valid}} = \mathbf{C}_{\text{full}}[1:2, 1:2]$, 显然, 这符合 full 卷积和 valid 卷积的关系。

2. full 卷积与 same 卷积的关系

假设 same 卷积的卷积核的锚点的位置在第 Fr 行、第 Fc 列处, 那么 \mathbf{C}_{full} 和 \mathbf{C}_{same} 的关系为

$$\mathbf{C}_{\text{same}} = \mathbf{C}_{\text{full}}[\text{FH} - \text{Fr} - 1 : H + \text{FH} - \text{Fr} - 2, \text{FW} - \text{Fc} - 1 : W + \text{FW} - \text{Fc} - 2]$$

仍以 8.1 节示例中的 \mathbf{x} 和 \mathbf{K} 为例, \mathbf{K} 的锚点的位置在第 $\text{Fr} = 0$ 行、第 $\text{Fc} = 0$ 列处, 有

$$\mathbf{C}_{\text{same}} = \mathbf{C}_{\text{full}}[1:3, 1:3]$$

即 \mathbf{C}_{full} 的第 1 行至第 3 行、第 1 列至第 3 列为 \mathbf{C}_{same} 。

3. same 卷积与 valid 卷积的关系

$\mathbf{C}_{\text{valid}}$ 和 \mathbf{C}_{same} 的关系为

$$\mathbf{C}_{\text{valid}} = \mathbf{C}_{\text{same}}[\text{Fr} : H - \text{FH} + \text{Fr}, \text{Fc} : W - \text{FW} + \text{Fc}]$$

仍以 8.1 节示例中的 \mathbf{x} 和 \mathbf{K} 为例, 有

$$\mathbf{C}_{\text{valid}} = \mathbf{C}_{\text{same}}[0:1, 0:1]$$

综合上述, 8.1 节示例中的 \mathbf{x} 和 \mathbf{K} 的三种卷积结果的关系如图 8-9 所示。

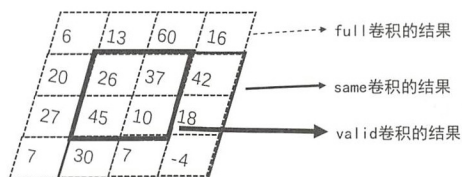


图 8-9 三种卷积类型的关系

8.1.5 卷积结果的输出尺寸

我们讨论的卷积操作, 在卷积过程中卷积核的移动步长均是 1, 所以 H 行 W 列的 \mathbf{x} 与 FH 行 FW 列的卷积核 \mathbf{K} 的 same 卷积结果的尺寸为 H 行 W 列, valid 卷积结果的尺寸为 $H - \text{FH} + 1$ 行 $W - \text{FW} + 1$ 列。

仍以 8.1 节的 \mathbf{x} 和 \mathbf{K} 为例，其中 \mathbf{x} 的尺寸为 3 行 3 列， \mathbf{K} 的尺寸为 2 行 2 列，图 8-8 所示为两者 valid 卷积的结果，其尺寸为 $3-2+1$ 行 $3-2+1$ 列，即为 2 行 2 列。

至此，我们讨论的卷积过程中卷积核的移动步长均为 1。接着，我们讨论常见的情况，假设有 H 行 W 列的输入张量 \mathbf{x} ， FH 行 FW 列的卷积核 \mathbf{K} ，在两者卷积的过程中，在垂直方向上，卷积核的移动步长为 SH ；在水平方向上，卷积核的移动步长为 SW ，如何计算两者 same 卷积和 valid 卷积结果的尺寸呢？

1. same 卷积结果的尺寸

\mathbf{x} 与 \mathbf{K} 的 valid 卷积结果的尺寸为 $\text{ceil}(\frac{H}{SH})$ 行、 $\text{ceil}(\frac{W}{SW})$ 列，其中函数 $\text{ceil}(x)$ 代表取不小于 x 的最小整数。

举例：假设输入张量 \mathbf{x} 的尺寸是 $H = 5$ 行、 $W = 5$ 列，卷积核 \mathbf{K} 的尺寸为 $FH = 2$ 、 $FW = 2$ 列，假设卷积过程中垂直方向的移动步长 $SH = 3$ ，水平方向的移动步长 $SW = 2$ ，则 same 卷积的尺寸为 $\text{ceil}(\frac{H}{SH}) = \text{ceil}(\frac{5}{3}) = 2$ 行， $\text{ceil}(\frac{W}{SW}) = \text{ceil}(\frac{5}{2}) = 3$ 列。

2. valid 卷积结果的尺寸

\mathbf{x} 与 \mathbf{K} 的 valid 卷积结果的尺寸为 $\text{floor}(\frac{H-FH}{SH})+1$ （等于 $\text{ceil}(\frac{H-FH+1}{SH})$ ）行， $\text{floor}(\frac{W-FW}{SW})+1$ （等于 $\text{ceil}(\frac{W-FW+1}{SW})$ ）列，其中 $\text{floor}(x)$ 代表取不大于 x 的最大整数。

举例：8.1 节示例中的 \mathbf{x} 和 \mathbf{K} ，两者 valid 卷积结果的尺寸为 $\text{floor}(\frac{H-FH}{SH})+1 = \text{floor}(\frac{5-2}{3})+1 = 2$ 行， $\text{floor}(\frac{W-FW}{SW})+1 = \text{floor}(\frac{5-2}{2})+1 = 2$ 列。

如无特殊说明，后面章节默认卷积的移动步长为 1。

8.2 离散卷积的性质

8.2.1 可分离的卷积核

如果一个卷积核由至少两个尺寸比它小的卷积核 full 卷积而成，即满足

$$\mathbf{Kernel} = \mathbf{kernel}_1 \star \mathbf{kernel}_2 \star \cdots \mathbf{kernel}_n$$

其中 \mathbf{kernel}_i 的尺寸均比 \mathbf{Kernel} 小， $1 \leq i \leq n$ ，则称卷积核 \mathbf{Kernel} 是可分离的。

如图 8-10 所示，2 个 3 行 3 列的卷积核均可以分离为 1 行 3 列和 3 行 1 列的卷积核。

$$\begin{array}{c}
 \begin{array}{|c|c|c|} \hline 1 & 0 & -1 \\ \hline 1 & 0 & -1 \\ \hline 1 & 0 & -1 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline 1 & 0 & -1 \\ \hline \end{array} \star \begin{array}{|c|} \hline 1 \\ \hline 1 \\ \hline 1 \\ \hline \end{array} = \begin{array}{|c|} \hline 1 \\ \hline 1 \\ \hline 1 \\ \hline \end{array} \star \begin{array}{|c|c|c|} \hline -1 & 0 & 1 \\ \hline \end{array} \\
 \\
 \begin{array}{|c|c|c|} \hline 1 & 2 & 1 \\ \hline 0 & 0 & 0 \\ \hline -1 & -2 & -1 \\ \hline \end{array} = \begin{array}{|c|} \hline 1 \\ \hline 0 \\ \hline -1 \\ \hline \end{array} \star \begin{array}{|c|c|c|} \hline 1 & 2 & 1 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline 1 & 2 & 1 \\ \hline \end{array} \star \begin{array}{|c|} \hline -1 \\ \hline 0 \\ \hline 1 \\ \hline \end{array}
 \end{array}$$

图 8-10 分离的卷积核

8.2.2 full 和 same 卷积的性质

假设 I 与卷积核 \mathbf{Kernel} 卷积，其中 \mathbf{Kernel} 是可分离的，即

$$\mathbf{Kernel} = \mathbf{kernel}_1 \star \mathbf{kernel}_2$$

full 卷积满足以下性质：

$$I \star \mathbf{Kernel} = I \star (\mathbf{kernel}_1 \star \mathbf{kernel}_2) = (I \star \mathbf{kernel}_1) \star \text{rotate180}(\mathbf{kernel}_2)$$

same 卷积满足以下性质：

$$I \star \mathbf{Kernel} = I \star (\mathbf{kernel}_1 \star \mathbf{kernel}_2) = (I \star \mathbf{kernel}_1) \star \text{rotate180}(\mathbf{kernel}_2)$$

我们可以通过以下示例理解该性质，输入张量和卷积核如图 8-11 所示。

$$\begin{array}{c}
 I = \begin{array}{|c|c|c|c|c|} \hline 2 & 9 & 11 & 4 & 8 \\ \hline 6 & 12 & 20 & 16 & 5 \\ \hline 1 & 32 & 13 & 14 & 10 \\ \hline 11 & 20 & 27 & 40 & 17 \\ \hline 9 & 8 & 11 & 4 & 1 \\ \hline \end{array} \quad \mathbf{Kernel} = \begin{array}{|c|c|c|} \hline 4 & 8 & 12 \\ \hline 5 & 10 & 15 \\ \hline 6 & 12 & 18 \\ \hline \end{array}
 \end{array}$$

图 8-11 输入张量和卷积核

其中卷积核是可分离的，如图 8-12 所示。

$$\mathbf{Kernel} = \begin{array}{|c|c|c|} \hline 4 & 8 & 12 \\ \hline 5 & 10 & 15 \\ \hline 6 & 12 & 18 \\ \hline \end{array} = \begin{array}{|c|} \hline 4 \\ \hline 5 \\ \hline 6 \\ \hline \end{array} \star \begin{array}{|c|c|c|} \hline 3 & 2 & 1 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline 1 & 2 & 3 \\ \hline \end{array} \star \begin{array}{|c|} \hline 6 \\ \hline 5 \\ \hline 4 \\ \hline \end{array}$$

图 8-12 分离的卷积核

以下代码实现了 I 与卷积核 \mathbf{Kernel} 的 same 卷积，因为 \mathbf{Kernel} 是可分离的，利用 same 卷积的性质，可以计算两者的 same 卷积：

图解深度学习与神经网络：从张量到 TensorFlow 实现

```

# -*- coding: utf-8 -*-
import tensorflow as tf
#"输入张量5×5"
I=tf.constant(
    [
        [
            [[2],[9],[11],[4],[8]],
            [[6],[12],[20],[16],[5]],
            [[1],[32],[13],[14],[10]],
            [[11],[20],[27],[40],[17]],
            [[9],[8],[11],[4],[1]]
        ]
    ],tf.float32
)
#"卷积核3×3"
Kernel=tf.constant(
    [
        [[4]],[8]],[[12]]],
        [[5]],[[10]],[[15]]],
        [[6]],[[12]],[[18]]]
    ],tf.float32
)
#"创建会话"
session=tf.Session()
#"输入张量与卷积核直接卷积"
result=tf.nn.conv2d(I,Kernel,[1,1,1,1],'SAME')
print('直接卷积的结果')
print(session.run(result))
#"卷积核可以分离为3×1的垂直卷积核和1×3的水平卷积核"
kernel1=tf.constant(
    [
        [[4]]],
        [[5]]],
        [[6]]]
    ],tf.float32
)

```

```

kernel2=tf.constant(
    [
        [[3]], [[2]], [[1]]
    ],tf.float32
)
#"将kernel2翻转180°"
rotate180_kernel2=tf.reverse(kernel2,axis=[1])
#"输入张量与分离的卷积核的卷积"
result1=tf.nn.conv2d(I,kernel1,[1,1,1,1],'SAME')
result12=tf.nn.conv2d(result1,rotate180_kernel2,[1,1,1,1],'SAME')
print('利用卷积核的分离性的卷积结果')
print(session.run(result12))

```

上述程序的打印结果如下：

```

"直接卷积的结果"
[[[ 443.], [ 805.], [ 815.], [ 617.], [ 256.]],
 [ 952.], [1286.], [1272.], [ 933.], [ 414.]],
 [1174.], [1672.], [2064.], [1571.], [ 718.]],
 [1054.], [1424.], [1622.], [1206.], [ 542.]]
 [ 538.], [ 818.], [ 986.], [ 742.], [ 326.]]]]
"利用卷积核的分离性的卷积结果"
[[[ 443.], [ 805.], [ 815.], [ 617.], [ 256.]],
 [ 952.], [1286.], [1272.], [ 933.], [ 414.]],
 [1174.], [1672.], [2064.], [1571.], [ 718.]],
 [1054.], [1424.], [1622.], [1206.], [ 542.]],
 [ 538.], [ 818.], [ 986.], [ 742.], [ 326.]]]]

```

显然，通过直接卷积的方式计算的结果和利用卷积核的分离性计算的结果相等。接下来，我们介绍如果卷积核是可分离的，那么利用卷积核的分离性计算卷积的优势是什么？

8.2.3 快速计算卷积

假设输入张量 I 的尺寸是 $H \times W$ ，卷积核 **Kernel** 的尺寸为 $FH \times FW$ ，则两者 same 卷积的乘法计算次数为

$$H \times W \times FH \times FW$$

如果卷积核 **Kernel** 是可分离的，分离为 $FH \times 1$ 的垂直卷积核 \mathbf{kernel}_1 和 $1 \times FW$ 的水平卷积核 \mathbf{kernel}_2 ，则 $(I \star \mathbf{kernel}_1) \star \text{rotate180}(\mathbf{kernel}_2)$ 的乘法计算次数为

$$H \times W \times (FH + FW)$$

在 8.2.2 节的示例中，两者 same 卷积的计算次数为 $5 \times 5 \times 3 \times 3 = 225$ ，利用卷积核的分离性及卷积的结合率，same 卷积的计算次数为 $(5 \times 5) \times (3 + 3) = 150$ 。显然，利用卷积核的分离性，计算次数比直接卷积减少了很多，张量或者卷积核的尺寸越大，优势越明显。

8.3 节会将一维卷积定理推广到二维卷积，介绍如何通过二维离散傅里叶变换计算二维离散卷积。

8.3 二维卷积定理

二维卷积定理是第 7 章介绍的一维卷积定理的推广，它揭示了二维傅里叶变换和二维卷积的某种关系，接下来先介绍二维离散傅里叶变换，再引出二维卷积定理。

8.3.1 二维离散傅里叶变换

同一维离散傅里叶变换类似，假设有 M 行 N 列的复数数列 f ，其中 $f(x, y)$ 代表 f 第 x 行第 y 列对应的值，那么对任意的 $x \in [0, M-1]$ ， $y \in [0, N-1]$ ，是否存在 M 行 N 列的复数数列 F ，使得以下等式成立：

$$f(x, y) = \frac{1}{MN} \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} F(u, v) e^{(\frac{2\pi}{M}ux + \frac{2\pi}{N}vy)i}, \quad 0 \leq x < M, 0 \leq y < N$$

答案是肯定的。我们试着求解该复数数列 F 。

解：对任意固定的 u_1 和 v_1 ，其中 $0 \leq u_1 < M$ ， $0 \leq v_1 < N$ ，上述等式两边同乘以 $e^{-(\frac{2\pi}{M}u_1x + \frac{2\pi}{N}v_1y)i}$ ，即

$$\begin{aligned} MN \times f(x, y) e^{-(\frac{2\pi}{M}u_1x + \frac{2\pi}{N}v_1y)i} = \\ F(0, 0) e^{-(\frac{2\pi}{M}(0-u_1)x + \frac{2\pi}{N}(0-v_1)y)i} + \dots + F(0, N-1) e^{-(\frac{2\pi}{M}(0-u_1)x + \frac{2\pi}{N}((N-1)-v_1)y)i} + \\ F(1, 0) e^{-(\frac{2\pi}{M}(1-u_1)x + \frac{2\pi}{N}(0-v_1)y)i} + \dots + F(1, N-1) e^{-(\frac{2\pi}{M}(1-u_1)x + \frac{2\pi}{N}((N-1)-v_1)y)i} + \\ F(2, 0) e^{-(\frac{2\pi}{M}(2-u_1)x + \frac{2\pi}{N}(0-v_1)y)i} + \dots + F(2, N-1) e^{-(\frac{2\pi}{M}(2-u_1)x + \frac{2\pi}{N}((N-1)-v_1)y)i} + \\ F(3, 0) e^{-(\frac{2\pi}{M}(3-u_1)x + \frac{2\pi}{N}(0-v_1)y)i} + \dots + F(3, N-1) e^{-(\frac{2\pi}{M}(3-u_1)x + \frac{2\pi}{N}((N-1)-v_1)y)i} + \end{aligned}$$

+ +

$$\mathbf{F}(M-1, 0)e^{-\left(\frac{2\pi}{M}((M-1)-u_1)x + \frac{2\pi}{N}(0-v_1)y\right)i} + \dots + \mathbf{F}(M-1, N-1)e^{-\left(\frac{2\pi}{M}((M-1)-u_1)x + \frac{2\pi}{N}((N-1)-v_1)y\right)i}$$

接着, 在上述等式两边分别针对 x 和 y 求和, 则

$$\begin{aligned} MN \times \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x, y) e^{-\left(\frac{2\pi}{M}u_1x + \frac{2\pi}{N}v_1y\right)i} = \\ \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} \mathbf{F}(0, 0) e^{-\left(\frac{2\pi}{M}(0-u_1)x + \frac{2\pi}{N}(0-v_1)y\right)i} + \dots + \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} \mathbf{F}(0, N-1) e^{-\left(\frac{2\pi}{M}(0-u_1)x + \frac{2\pi}{N}((N-1)-v_1)y\right)i} + \\ \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} \mathbf{F}(1, 0) e^{-\left(\frac{2\pi}{M}(1-u_1)x + \frac{2\pi}{N}(0-v_1)y\right)i} + \dots + \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} \mathbf{F}(1, N-1) e^{-\left(\frac{2\pi}{M}(1-u_1)x + \frac{2\pi}{N}((N-1)-v_1)y\right)i} + \\ \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} \mathbf{F}(2, 0) e^{-\left(\frac{2\pi}{M}(2-u_1)x + \frac{2\pi}{N}(0-v_1)y\right)i} + \dots + \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} \mathbf{F}(2, N-1) e^{-\left(\frac{2\pi}{M}(2-u_1)x + \frac{2\pi}{N}((N-1)-v_1)y\right)i} + \\ \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} \mathbf{F}(3, 0) e^{-\left(\frac{2\pi}{M}(3-u_1)x + \frac{2\pi}{N}(0-v_1)y\right)i} + \dots + \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} \mathbf{F}(3, N-1) e^{-\left(\frac{2\pi}{M}(3-u_1)x + \frac{2\pi}{N}((N-1)-v_1)y\right)i} + \\ + \dots \dots + \\ \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} \mathbf{F}(M-1, 0) e^{-\left(\frac{2\pi}{M}((M-1)-u_1)x + \frac{2\pi}{N}(0-v_1)y\right)i} + \dots + \\ \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} \mathbf{F}(M-1, N-1) e^{-\left(\frac{2\pi}{M}((M-1)-u_1)x + \frac{2\pi}{N}((N-1)-v_1)y\right)i} \end{aligned}$$

只要满足 $(u-u_1)$ 和 $(v-v_1)$ 不同时为 0, 则

$$\sum_{x=0}^{M-1} \sum_{y=0}^{N-1} e^{-\left(\frac{2\pi}{M}(u-u_1)x + \frac{2\pi}{N}(v-v_1)y\right)i} = \sum_{x=0}^{M-1} e^{-\left(\frac{2\pi}{M}(u-u_1)i\right)x} \sum_{y=0}^{N-1} e^{-\left(\frac{2\pi}{N}(v-v_1)i\right)y} = 0$$

所以

$$MN \times \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x, y) e^{-\left(\frac{2\pi}{M}u_1x + \frac{2\pi}{N}v_1y\right)i} = MN \times \mathbf{F}(u_1, v_1)$$

即

$$\mathbf{F}(u_1, v_1) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x, y) e^{-\left(\frac{2\pi}{M}u_1x + \frac{2\pi}{N}v_1y\right)i}$$

因为上式是对任意的 u_1 和 v_1 计算出来的结果，所以把 u_1 和 v_1 换成 u 和 v ，即可得到

$$\mathbf{F}(u, v) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x, y) e^{-j(\frac{2\pi}{M}ux + \frac{2\pi}{N}vy)}, \quad 0 \leq u < M, 0 \leq v < N$$

那么 \mathbf{F} 称为 f 的傅里叶变换，而 f 称为 \mathbf{F} 的傅里叶逆变换，表示为 $f \Leftrightarrow \mathbf{F}$ 。

TensorFlow 通过函数 `fft2d` 和 `ifft2d` 实现二维离散的傅里叶变换及逆变换，我们利用这两个函数计算图 8-13 所示的二维矩阵的傅里叶变换。

10	2	8
5	12	3

图 8-13 二维矩阵

具体实现代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
#"二维张量"
f=tf.constant([
    [10,2,8],
    [5,12,3]
],tf.complex64)
#"创建会话"
session=tf.Session()
#"f的二维离散傅里叶变换"
F=tf.fft2d(f)
#"打印f的傅里叶变换的值"
print("f的二维离散傅里叶变换:")
print(session.run(F))
#"计算F的傅里叶逆变换(显然与输入的f是相等的)"
F_ifft2d=tf.ifft2d(F)
#"打印F的傅里叶逆变换的值"
print("F的傅里叶逆变换:")
print(session.run(F_ifft2d))
```

打印结果如下：

"f的二维离散傅里叶变换:"

```
[[ 4.000e+01 -8.344e-07j 2.500e+00 -2.59807348e+00j 2.499e+00 +2.598e+00j]
 [ 9.536e-07 +1.192e-06j 7.500e+00 +1.29903812e+01j 7.499e+00 -1.299e+01j]]
```

"F的傅里叶逆变换:"

```
[[ 10.000 +4.768e-07j 2.000 -6.357e-07j 8.000 +0.000e+00j]
 [ 5.000 +1.430e-06j 12.000 -3.178e-07j 3.000 -1.748e-06j]]
```

从以上程序的打印结果可以得出如图 8-14 所示的结论。

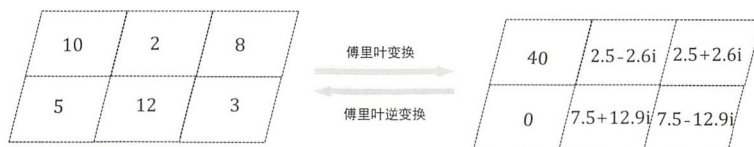


图 8-14 傅里叶（逆）变换

8.3.2 二维与一维傅里叶变换的关系

二维离散傅里叶变换可以分解为一维离散傅里叶变换：

$$\mathbf{F}(u, v) = \sum_{x=0}^{M-1} \left[\sum_{y=0}^{N-1} f(x, y) e^{-\frac{2\pi}{N} v y i} \right] e^{-\frac{2\pi}{M} u x i}, \quad 0 \leq u < M, 0 \leq v < N$$

方括号中的项表示在行方向上计算傅里叶变换，方括号外的求和代表在行的傅里叶变换的基础上，接着计算列方向上的傅里叶变换。同理，二维离散傅里叶变换也可以分解为先计算每一列的傅里叶变换，再计算每一行的傅里叶变换，通过以下示例可以更好地理解这句话。

图 8-13 所示的二维矩阵的二维傅里叶变换可以分 2 步完成：先分别计算每一列的一维傅里叶变换，再计算每一行的一维傅里叶变换，过程如图 8-15 所示。

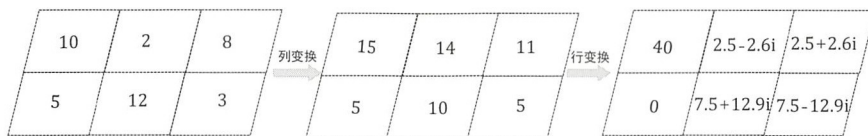


图 8-15 先列变换，然后行变换

当然，也可以先计算每一行的一维傅里叶变换，再计算每一列的一维傅里叶变换。

TensorFlow 并没有直接提供分别计算二维数列的行或列的傅里叶变换，Numpy 中的函数 `fft` 可以实现该功能，具体代码如下：

图解深度学习与神经网络：从张量到 TensorFlow 实现

```
# -*- coding: utf-8 -*-
import numpy as np
f=np.array([
    [10,2,8],
    [5,12,3]
],np.complex64)
#"第1步：对每一列进行傅里叶变换"
f_0_fft=np.fft.fft(f,axis=0)
print(f_0_fft)
#"第2步：针对第1步得到的结果，分别对每一行进行傅里叶变换"
f_0_1_fft=np.fft.fft(f_0_fft,axis=1)
print(f_0_1_fft)
```

打印结果如下：

```
[[ 40.0 +0.j           2.5 -2.59807621j   2.5 +2.59807621j]
 [  0.0 +0.j           7.5+12.99038106j   7.5-12.99038106j]]
```

以下代码是先计算每一行的一维傅里叶变换，再计算每一列的一维离散傅里叶变换，代码如下：

```
#"第1步：对每一行进行傅里叶变换"
f_1_fft=np.fft.fft(f,axis=1)
print(f_1_fft)
"第2步：针对第1步得到的结果，分别对每一列进行傅里叶变换"
f_1_0_fft=np.fft.fft(f_1_fft,axis=0)
print(f_1_0_fft)
```

打印结果如下：

```
[[ 40.0 +0.j           2.5 -2.59807621j   2.5 +2.59807621j]
 [  0.0 +0.j           7.5+12.99038106j   7.5-12.99038106j]]
```

我们可以发现，打印的结果相等，均是最后的二维离散傅里叶变换的结果。

更高维数的傅里叶变换的原理同一维和二维类似。傅里叶变换在第9章将介绍的卷积运算的快速算法中发挥着重要的作用。

8.3.3 卷积定理

假设有高为 H 、宽为 W 的二维输入张量 I ，高为 FH 、宽为 FW 的卷积核 k ，那么 I 与 k 的 full 卷积结果的尺寸是高为 $H + FH - 1$ 、宽为 $W + FW - 1$ 。

在 I 的右侧和下层补零，且将 I 的尺寸扩充到与 full 卷积的尺寸相同，即

$$I_{\text{padded}}(h, w) = \begin{cases} I(h, w), & 0 \leq h < H, 0 \leq w < W \\ 0, & \text{其他} \end{cases}$$

其中 $0 \leq h < H + FH - 1$, $0 \leq w < W + FW - 1$ 。

将卷积核 k 逆时针翻转 180° 得到 $k_{\text{rotate180}}$ ，然后对其右侧和下侧进行补零，且将 $k_{\text{rotate180}}$ 的尺寸扩充到和 full 卷积同样的尺寸，即

$$k_{\text{rotate180_padded}}(fh, fw) = \begin{cases} k_{\text{rotate180}}(fh, fw), & 0 \leq fh < FH, 0 \leq fw < FW \\ 0, & \text{其他} \end{cases}$$

其中 $0 \leq h < H + FH - 1$, $0 \leq w < W + FW - 1$ 。

假设 fft2_Ip 和 fft2_krp 分别是 I_{padded} 和 $k_{\text{rotate180_padded}}$ 的傅里叶变换，那么 $I \star k$ 的傅里叶变换等于 $\text{fft2_Ip} * \text{fft2_krp}$ ，即

$$I \star k \Leftrightarrow \text{fft2_Ip} * \text{fft2_krp}$$

其中 $*$ 代表对应位置的元素相乘，即对应位置的两个复数相乘，该性质称为卷积定理。

8.3.4 利用卷积定理快速计算卷积

我们以 8.1 节示例中的 x 和 K 为例，利用卷积定理计算两者的卷积，具体实现代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
#"输入张量I"
I=tf.constant(
    [
        [2,3,8],
        [6,1,5],
        [7,2,-1]
    ],tf.complex64
```

```
)
#"卷积核"
k=tf.constant(
    [
        [4,1],
        [2,3]
    ],tf.complex64
)
#"对输入张量的下侧和右侧补0"
I_padded=tf.pad(I,[[0,1],[0,1]])
#"翻转卷积核180°"
k_rotate180=tf.reverse(k,[0,1])
#"对翻转后的卷积核下侧和右侧补0"
k_rotate180_padded=tf.pad(k_rotate180,[[0,2],[0,2]])
#"二维离散傅里叶变换"
I_padded_fft2=tf.fft2d(I_padded)
k_rotate180_padded_fft2=tf.fft2d(k_rotate180_padded)
#"两个二维傅里叶变换对应位置相乘"
xk_fft2=tf.multiply(I_padded_fft2,k_rotate180_padded_fft2)
#"对以上相乘的结果进行傅里叶逆变换"
xk=tf.ifft2d(xk_fft2)
#"创建会话"
session=tf.Session()
#"利用卷积定理计算的full卷积的结果"
print(session.run(xk))
```

打印结果如下：

```
[[ 6.+0.j 13.+0.j 30.+0.j 16.+0.j]
 [20.+0.j 26.+0.j 37.+0.j 42.+0.j]
 [27.+0.j 45.+0.j 10.+0.j 18.+0.j]
 [ 7.+0.j 30.+0.j  7.+0.j -4.+0.j]]
```

打印的结果为两者 full 卷积的结果。

至此，介绍的卷积运算可以理解为深度为 1 的离散卷积，接下来我们介绍多深度张量的卷积运算。

8.4 多深度的离散卷积

8.4.1 基本的多深度卷积

我们以图 8-16 所示的 3 行 3 列 2 深度的三维张量 x 和 2 行 2 列 2 深度的三维卷积核 k 的 valid 卷积为例，介绍基本的多深度卷积。

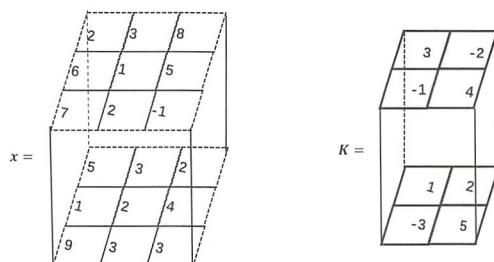


图 8-16 三维张量和三维卷积核

两者分别在每一深度上进行二维 valid 卷积，然后在深度方向求和，过程如图 8-17 所示。

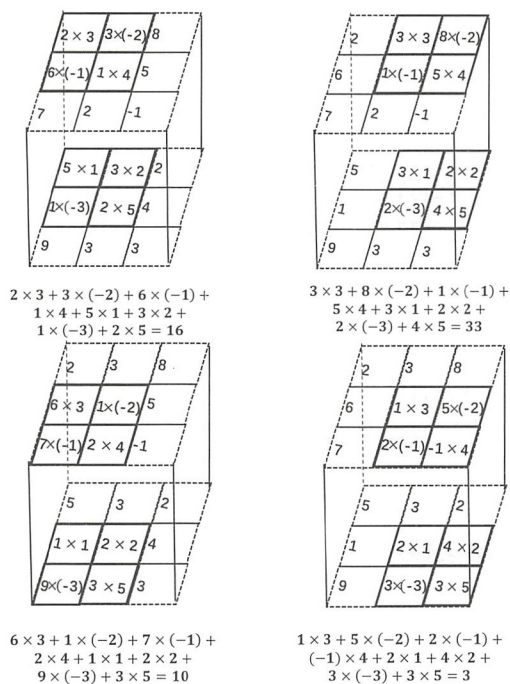
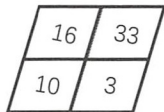


图 8-17 valid 卷积的过程

即两者的 valid 卷积的结果如图 8-18 所示。



16	33
10	3

图 8-18 valid 卷积的结果

上述过程的具体代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
#"1个3行2列2深度的张量"
x=tf.constant(
    [
        [
            [[2,5],[3,3],[8,2]],
            [[6,1],[1,2],[5,4]],
            [[7,9],[2,3],[-1,3]]
        ]
        ],tf.float32
    )
#"1个2行2列2深度的卷积核"
k=tf.constant(
    [
        [[3],[1]], [[-2],[2]],
        [[-1],[-3]], [[4],[5]]
    ],tf.float32
    )
#"每一深度分别计算卷积，然后求和"
x_conv2d_k=tf.nn.conv2d(x,k,[1,1,1,1],'VALID')
#"创建会话"
session=tf.Session()
#"打印结果"
print(session.run(x_conv2d_k))
```

打印结果如下：

```
[[[ 16.],[ 33.]],
 [[ 10.],[  3.]]]
```

即 1 个 3 行 3 列 2 深度的三维张量与 1 个 2 行 2 列 2 深度的卷积核的 valid 卷积，结果是 1 个 2 行 2 列深度 1 的三维张量。

接下来介绍 1 个三维张量与多个卷积核的卷积，这些卷积核的深度和输入的三维张量的深度必须相等。

8.4.2 1 个张量与多个卷积核的卷积

我们通过图 8-19 所示的示例来理解 1 个张量与多个卷积核的卷积，1 个 3 行 3 列 2 深度的张量与 3 个 2 行 2 列 2 深度的卷积核卷积。

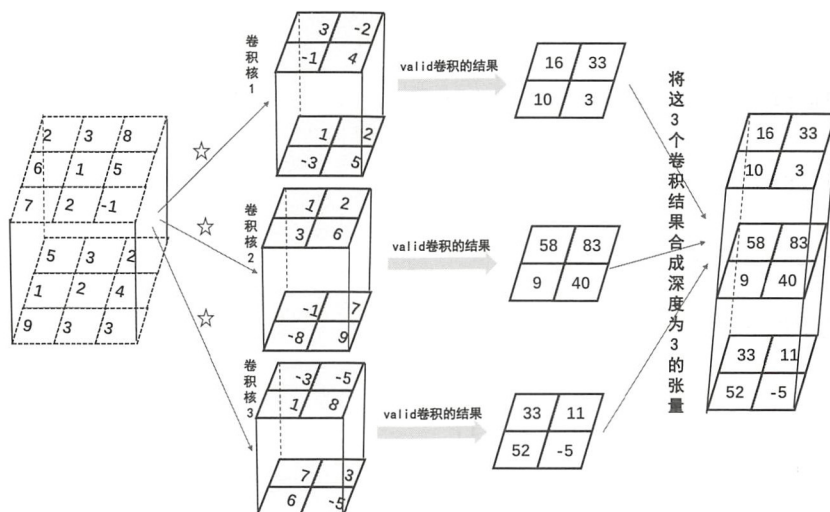


图 8-19 1 个张量与多个卷积核的卷积

从卷积过程可知，该张量分别与每个卷积核进行基本的多深度卷积，然后将得到的每个结果在深度上连接，该过程仍用函数 `tf.nn.conv2d` 实现，具体代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
inputValue=tf.constant(
    [
        #"1个深度是2且有3行3列的张量"
        [
            [[2,5],[3,3],[8,2]],
            [[6,1],[1,2],[5,4]],
        ]
    ]
)
```



```
        [[7,9],[2,3],[-1,3]]
    ]
    ],tf.float32
)

#"3个深度是2且为2行2列的卷积核"
kernels=tf.constant(
    [
        [[3,1,-3],[1,-1,7]], [[-2,2,-5],[2,7,3]],
        [[-1,3,1],[-3,-8,6]], [[4,6,8],[5,9,-5]]
    ],tf.float32
)

#"valid卷积"
validResult=tf.nn.conv2d(inputValue,kernels,[1,1,1,1],'VALID')

#"创建会话"
session=tf.Session()

#"打印结果"
print(session.run(validResult))
```

打印结果如下：

```
[[
  [[ 16.  58.  33.],[ 33.  83.  11.]],
  [[ 10.   9.  52.],[   3.  40. -5.]]
]]
```

即 1 个 3 行 3 列 2 深度的输入张量，与 3 个 2 行 2 列 2 深度的卷积核的 valid 卷积结果是 1 个 2 行 2 列 3 深度的三维张量。

我们利用函数 `tf.nn.conv2d` 计算了 1 个张量与多个卷积核的卷积，其实该函数最大的优点是可以实现任意多个张量分别与多个卷积核的卷积。

8.4.3 多个张量分别与多个卷积核的卷积

我们通过图 8-20 所示的示例理解多个张量与多个卷积核的卷积，其中有 2 个深度为 2 的三维张量，分别与 3 个深度为 2 的卷积核进行基本的多深度卷积，需要注意的是每一个卷积核的深度必须与输入张量的深度相等，则第 1 个三维张量与 3 个深度为 2 的卷积核的多深

度卷积结果是深度为 3 的三维张量，第 2 个三维张量与 3 个深度为 2 的卷积核的多深度卷积的结果同样是深度为 3 的三维张量。

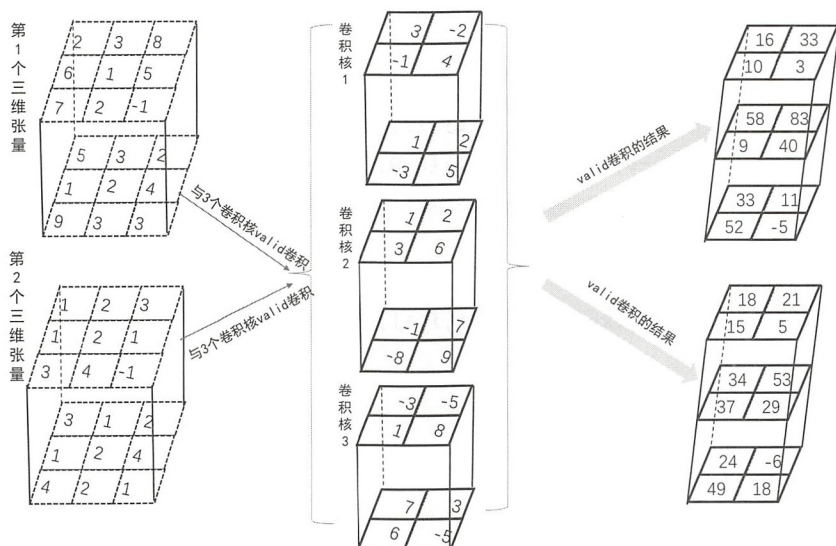


图 8-20 多个张量分别与多个卷积核的卷积

利用函数 `tf.nn.conv2d` 实现上述示例的具体代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
#"输入张量"
value=tf.constant(
    [
        #"第1个3行3列2深度的三维张量"
        [
            [[2,5],[3,3],[8,2]],
            [[6,1],[1,2],[5,4]],
            [[7,9],[2,3],[-1,3]]
        ],
        #"第2个3行3列2深度的三维张量"
        [
            [[1,3],[2,1],[3,2]],
            [[1,1],[2,2],[1,4]],
            [[3,4],[4,2],[-1,1]]
        ]
    ]
)
```

```
        ],tf.float32
    )
#"3个2行2列2深度的卷积核"
kernels=tf.constant(
    [
        [[3,1,-3],[1,-1,7]], [[-2,2,-5],[2,7,3]],
        [[-1,3,1],[-3,-8,6]], [[4,6,8],[5,9,-5]]
    ],tf.float32
)
#"valid卷积"
result=tf.nn.conv2d(value,kernels,[1,1,1,1],'VALID')
#"创建会话"
session=tf.Session()
#"打印结果"
print(session.run(result))
```

以上程序的打印结果如下：

```
[
#"第1个张量与3个卷积核的卷积结果"
[[[ 16.  58.  33.]
   [ 33.  83.  11.]]

 [[ 10.   9.  52.]
   [  3.  40. -5.]]],
#"第2个张量与3个卷积核的卷积结果"
[[[ 18.  34.  24.]
   [ 21.  53. -6.]]

 [[ 15.  37.  49.]
   [  5.  29.  18.]]]
]
```

以上代码利用函数 `tf.nn.conv2d` 计算了 2 个 3 行 3 列 2 深度的输入张量分别与 3 个 2 行 2 列 2 深度的卷积核的卷积，其结果为 2 个 2 行 2 列 3 深度的三维张量（即四维张量）。

总结：利用函数 `tf.nn.conv2d` 可以计算 M 个深度为 D 三维张量分别与 N 个深度为 D 的卷积核的卷积，其返回结果为 M 个深度为 N 的三维张量（即四维张量）。

函数 `tf.nn.conv2d` 实现的是分别在深度上卷积，然后沿深度上求和的卷积计算方式。接下来介绍另一个函数 `depthwise_conv2d`，该函数实现的只是在深度上卷积。

8.4.4 在每一深度上分别卷积

仍以 8.4.1 节中的两个张量为例，函数 `depthwise_conv2d` 实现的 valid 卷积的计算过程如图 8-21 所示。

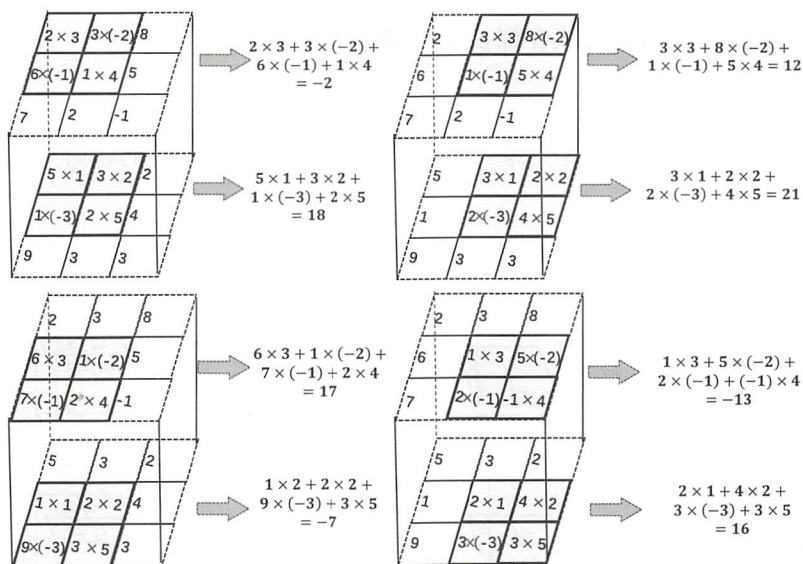


图 8-21 在每一深度上分别卷积的过程

其结果如图 8-22 所示。

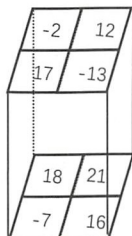


图 8-22 在每一深度上分别卷积的结果

函数 `depthwise_conv2d` 与函数 `conv2d` 的不同之处在于 `conv2d` 在每一深度上卷积，然后求和，`depthwise_conv2d` 没有求和这一步，上述过程的代码只需要将 8.4.3 节代码中的函

数 conv2d 替换成 depthwise_conv2d 即可，具体代码如下：

```
x_depthwise_conv2d_k=tf.nn.depthwise_conv2d(x,k,[1,1,1,1], 'VALID')
```

打印结果如下：

```
[[[[-2,18.],[ 12,21.]],  
 [[ 17,-7.],[-13,16.]]]]
```

8.4.5 单个张量与多个卷积核在深度上分别卷积

8.4.4 节实现的是 1 个三维张量与 1 个同等深度的卷积核的卷积，本节将介绍如何使用函数 depthwise_conv2d 计算 1 个三维张量与多个同等深度的卷积核的卷积，示例如图 8-23 所示。

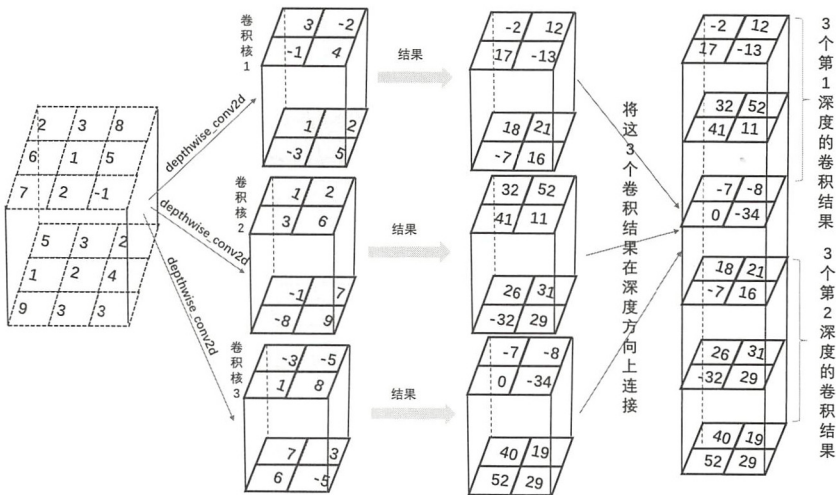


图 8-23 单个张量与多个卷积核在深度上分别卷积

1 个 3 行 3 列 2 深度的三维张量与 3 个 2 行 2 列 2 深度的三维卷积核卷积，因为输入张量与每个卷积核的卷积结果的深度为 2，一共与 3 个卷积核卷积，即有 3 个卷积结果，将它们在深度方向上连接，所以最终结果的深度为 $3 \times 2 = 6$ ，具体代码如下：

```
# -*- coding: utf-8 -*-  
import tensorflow as tf  
#"1个3行2列2深度的张量"  
x=tf.constant(  

```

```

        [
        [
            [[2,5],[3,3],[8,2]],
            [[6,1],[1,2],[5,4]],
            [[7,9],[2,3],[-1,3]]
        ]
        ],tf.float32
    )
    #"3个2行2列2深度的卷积核"
    k=tf.constant(
        [
            [[[3,1,-3],[1,-1,7]], [[-2,2,-5],[2,7,3]]],
            [[[-1,3,1],[-3,-8,6]], [[4,6,8],[5,9,-5]]]
        ],tf.float32
    )
    #
    #"每一深度分别计算卷积"
    x_depthwise_conv2d_k=tf.nn.depthwise_conv2d(x,k,[1,1,1,1], 'VALID')
    #"创建会话"
    session=tf.Session()
    #"打印结果"
    print(session.run(x_depthwise_conv2d_k))

```

打印结果如下：

```

[[[[-2.  32.  -7.  18.  26.  40.]
  [ 12.  52.  -8.  21.  31.  19.]]

  [[ 17.  41.   0.  -7. -32.  52.]
  [-13.  11. -34.  16.  29.  29.]]]]

```

总结：1 个深度为 D 的三维张量与 N 个深度为 D 的卷积核的 `depthwise_conv2d` 卷积，其结果为 1 个深度为 $N \times D$ 的三维张量。

同函数 `tf.nn.conv2d` 类似，函数 `depthwise_conv2d` 也可以计算多个三维张量，同时分别与多个卷积核的 `depthwise_conv2d` 卷积，即 M 个深度为 D 的三维张量分别与 N 个深度为 D 的卷积核的 `depthwise_conv2d` 卷积，结果为 M 个深度为 $N \times D$ 的三维张量（即四维张量）。

8.4.6 分离卷积

我们来介绍 TensorFlow 实现的另一个关于卷积的函数：

```
separable_conv2d(input, depthwise_filter, pointwise_filter, strides,
padding, rate=None, name=None, data_format=None)
```

它实现的功能可以用图 8-24 所示的示例表示，其卷积过程分为 2 步。

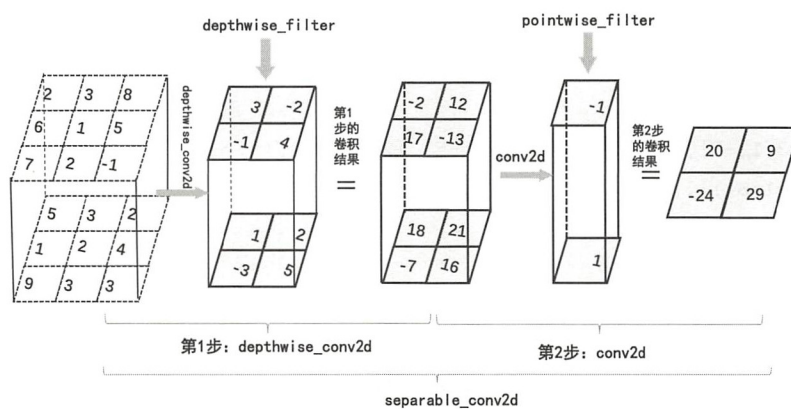


图 8-24 函数 separable_conv2d 的计算过程

函数 separable_conv2d 实现的功能是函数 depthwise_conv2d 和 conv2d 的组合，上述示例的代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
#"1个3行2列2深度的张量"
x=tf.constant(
    [
        [
            [[2,5],[3,3],[8,2]],
            [[6,1],[1,2],[5,4]],
            [[7,9],[2,3],[-1,3]]
        ],tf.float32
    )
#"1个2行2列2深度的卷积核depthwiseFilter"
depthwise_filter=tf.constant(
```

```

[
    [[3],[1]],[[-2],[2]]],
    [[[-1],[-3]],[[4],[5]]]
    ],tf.float32
)
#"1行1列2深度的卷积核pointwiseFilter"
pointwise_filter=tf.constant(
    [
        [[[-1],[1]]],
    ],tf.float32
)
#"分离卷积"
result=tf.nn.separable_conv2d(x,depthwise_filter,
                             pointwise_filter,[1,1,1,1],'VALID')
#"创建会话"
session=tf.Session()
#"打印结果"
print(session.run(result))

```

打印结果如下：

```

[[[ 20.],[ 9.]],
 [[-24.],[ 29.]]]

```

如图 8-25 所示，我们来考虑一个稍微复杂的分离卷积问题。

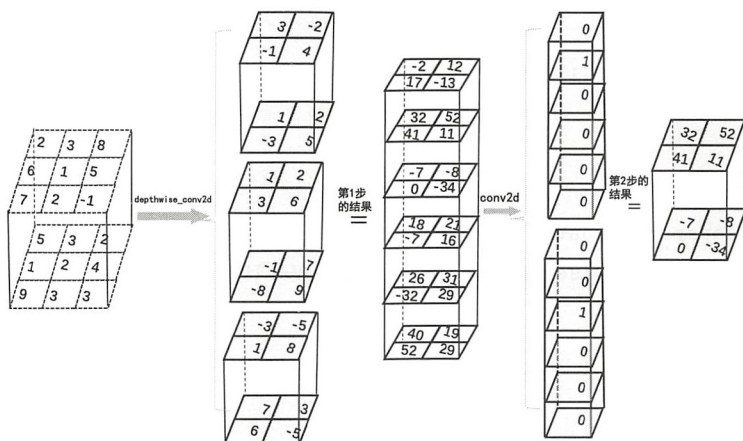


图 8-25 函数 `separable_conv2d` 实现的比较复杂的卷积

假设有 1 个 3 行 3 列 2 深度的三维张量，先与 3 个 2 行 2 列 2 深度的卷积核进行 `depthwise_conv2d` 卷积，其结果的深度为 6，然后与 2 个 1 行 1 列 6 深度的卷积核 `conv2d` 卷积，最后结果的深度为 2，具体实现代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
#"1个3行2列2深度的张量"
x=tf.constant(
    [
        [
            [[2,5],[3,3],[8,2]],
            [[6,1],[1,2],[5,4]],
            [[7,9],[2,3],[-1,3]]
        ]
    ],tf.float32
)
#"3个2行2列2深度的卷积核depthwiseFilter"
depthwise_filter=tf.constant(
    [
        [[3,1,-3],[1,-1,7]], [[-2,2,-5],[2,7,3]],
        [[-1,3,1],[-3,-8,6]], [[4,6,8],[5,9,-5]]
    ],tf.float32
)
#"2个1行1列6深度的卷积核pointwiseFilter"
pointwise_filter=tf.constant(
    [
        [[0,0],[1,0],[0,1],[0,0],[0,0],[0,0]],
    ],tf.float32
)
#"分离卷积"
result=tf.nn.separable_conv2d(x,depthwise_filter,
                             pointwise_filter,[1,1,1,1],'VALID')
#"创建会话"
session=tf.Session()
#"打印结果"
print(session.run(result))
```

打印结果如下:

```
[[[ 32.  -7.]  
  [ 52.  -8.]]  
  
[[ 41.   0.]  
 [ 11. -34.]]]
```

本章主要介绍了二维离散卷积的计算及其对应的性质,第9章将介绍另一种张量的操作:池化操作。

9

池化操作

pool（池化）操作与卷积运算类似，取输入张量的每一个位置的矩形邻域内值的最大值或者平均值作为该位置的输出值，如果取的是最大值，则称为**最大值池化**；如果取的是平均值，则称为**平均值池化**。pooling 操作在图像处理中的应用类似于均值平滑、形态学处理、下采样等操作，与卷积运算类似，池化操作也分为 same 池化和 valid 池化，下面我们依次介绍这两类池化操作。

9.1 same 池化

same 池化的操作方式一般有两种：same 最大值池化和 same 平均值池化。

9.1.1 same 最大值池化

我们以图 9-1 所示的 4 行 4 列的张量 \mathbf{x} 和 2 行 3 列的邻域掩码为例，介绍 same 最大值池化操作的计算过程。

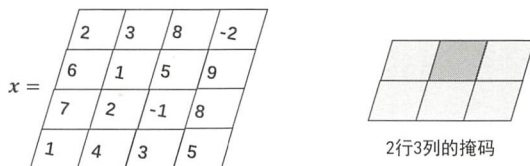


图 9-1 张量 \mathbf{x} 和邻域掩码

same 池化的邻域掩码同 same 卷积的卷积核类似，需要指定锚点，该锚点的位置和 same 卷积时卷积核的锚点位置类似，假设邻域掩码的高等于 FH、宽等于 FW，则：

- 如果 FH 为奇数，FW 为奇数，则锚点的位置是 $\left(\frac{FH-1}{2}, \frac{FW-1}{2}\right)$
- 如果 FH 为奇数，FW 为偶数，则锚点的位置是 $\left(\frac{FH-1}{2}, \frac{FW-2}{2}\right)$
- 如果 FH 为偶数，FW 为奇数，则锚点的位置是 $\left(\frac{FH-2}{2}, \frac{FW-1}{2}\right)$
- 如果 FH 为偶数，FW 为偶数，则锚点的位置是 $\left(\frac{FH-2}{2}, \frac{FW-2}{2}\right)$

这里的位置索引是从 0 开始的，所以 2 行 3 列邻域掩码锚点的位置在 (0, 1) 处，将锚点顺序移动到输入张量 \mathbf{x} 的每一个位置处，然后取掩码区域内的最大值，这就是 same 最大值池化操作的计算过程，如图 9-2 所示。

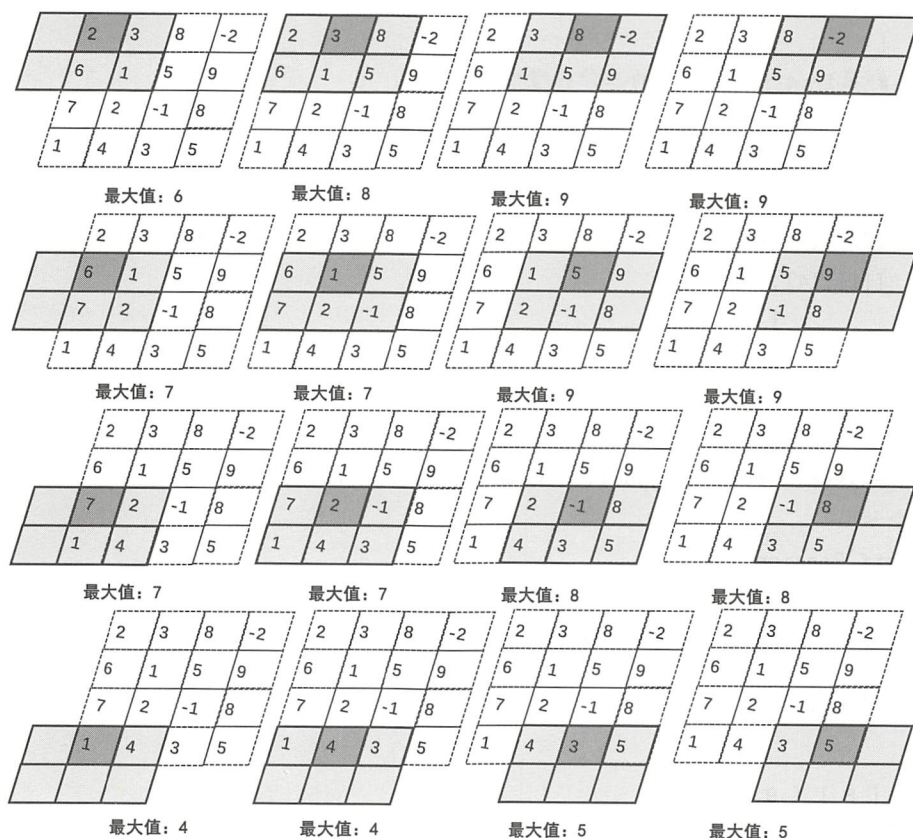


图 9-2 same 最大值池化的计算过程

将得到的值依次存入如图 9-3 所示的张量中，即两者最大值池化的结果。

6	8	9	9
7	7	9	9
7	7	8	8
4	4	5	5

图 9-3 最大值池化的结果

TensorFlow 通过函数 `tf.nn.max_pool` 实现最大值池化操作，以上示例的张量 `x` 可以看成 1 个 4 行 4 列 1 深度的三维张量，以上示例的 `same` 最大值池化操作的代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
#"输入形状为[1,4,4,1]的张量"
value2d = tf.constant(
    [
        #"第1个4行4列1深度的三维张量"
        [
            [[2],[3],[8],[-2]],
            [[6],[1],[5],[9]],
            [[7],[2],[-1],[8]],
            [[1],[4],[3],[5]]
        ],tf.float32
    )
#"最大值池化操作"
value2d_maxPool=tf.nn.max_pool(value2d,(1,2,3,1),[1,1,1,1],'SAME')
#"创建会话"
session=tf.Session()
#"打印结果"
print(session.run(value2d_maxPool))
```

打印结果如下：

```
[[
    [[ 6.],[ 8.],[ 9.],[ 9.]],
    [[ 7.],[ 7.],[ 9.],[ 9.]],
    [[ 7.],[ 7.],[ 8.],[ 8.]],
    [[ 4.],[ 4.],[ 5.],[ 5.]]
]]
```

上述代码处理的是深度为 1 的三维张量的池化操作，且邻域掩码在沿行和列方向的移动步长均为 1。接下来我们介绍多深度的三维张量的池化操作。

9.1.2 多深度张量的 same 池化

本质上，多深度的 same 最大值池化操作是在每一深度上分别进行池化操作，我们以图 9-4 所示的深度为 2 的三维张量和 2 行 2 列邻域掩码的 same 最大值池化操作为例介绍。

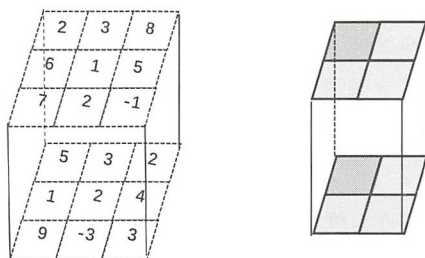


图 9-4 三维张量与邻域掩码

如果 same 池化操作过程中邻域掩码沿行和沿列的移动步长均为 2，如图 9-5 所示，那么此时两者 same 最大值池化的结果如图 9-6 所示。

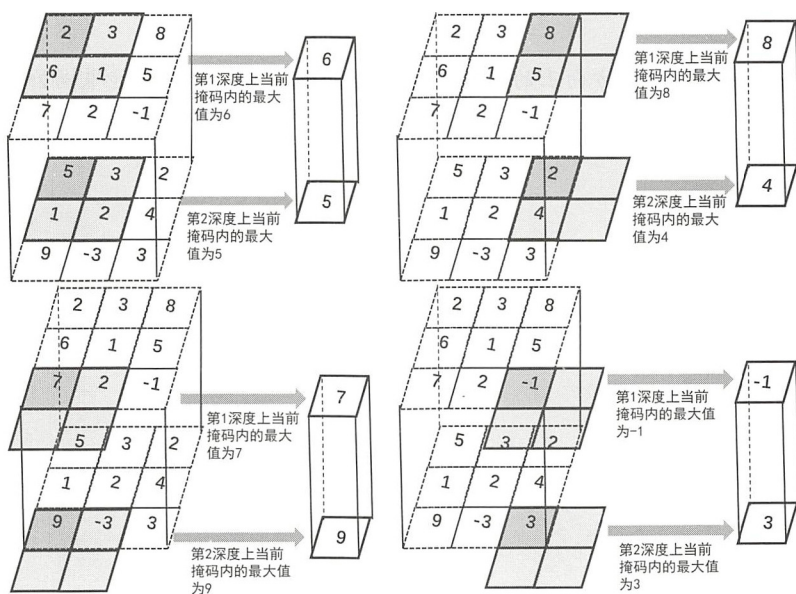


图 9-5 沿行和沿列的移动步长均为 2 的 same 最大值池化

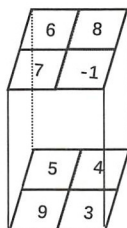


图 9-6 same 最大值池化的结果

本质上，多深度张量的 same 池化是在每一深度上分别池化操作，结果并未改变原张量的深度，以上示例对应的代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
value3d=tf.constant(
    [
        # "第1个3行3列2深度的三维张量"
        [
            [[2,5],[3,3],[8,2]],
            [[6,1],[1,2],[5,4]],
            [[7,9],[2,-3],[-1,3]]
        ]
    ],tf.float32
)
# "2行2列的池化掩码，在行方向上的移动步长为2，在列方向上的移动步长为2"
valued3d_maxPool=tf.nn.max_pool(value3d,[1,2,2,1],[1,2,2,1],'SAME')
# "创建会话"
session=tf.Session()
# "打印结果"
print(session.run(valued3d_maxPool))
```

打印结果如下：

```
[[[[ 6.  5.]
    [ 8.  4.]]

  [[ 7.  9.]
    [-1.  3.]]]]
```

显然， H 行 W 列张量的 same 池化操作，如果邻域掩码为 FH 行 FW 列，且在列方

向上的移动步长为 SW 、在行方向上的移动步长为 SH ，那么 same 池化的结果为 $\text{ceil}(\frac{H}{SH})$ 行、 $\text{ceil}(\frac{W}{SW})$ 列，其中 $\text{ceil}(x)$ 代表计算不大于 x 的最大整数。

9.1.3 多个三维张量的 same 最大值池化

以上使用最大值池化操作函数 `tf.nn.max_pool` 处理的都是一个三维张量，其实该函数可以同时计算任意多个三维张量的池化操作。

图 9-7 所示为两个深度为 2 的三维张量分别与 2 行 2 列邻域掩码的 same 最大值池化操作，其沿行和沿列方向的移动步长为 2。

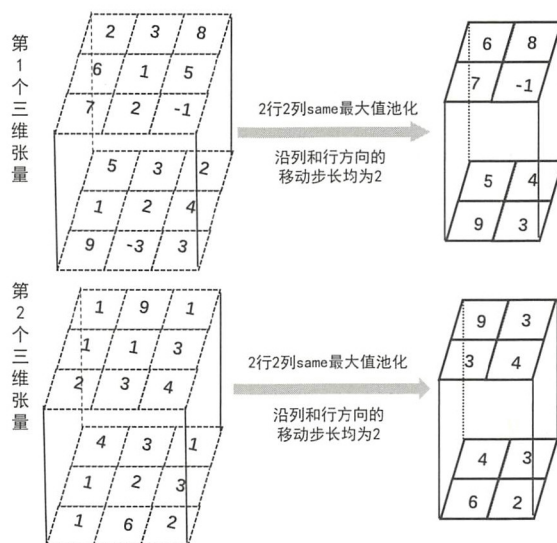


图 9-7 多个三维张量的 same 最大值池化结果

上述示例对应的具体代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
value3d=tf.constant(
    [
        #"第1个3行3列2深度的三维张量"
        [
            [[2,5],[3,3],[8,2]],
            [[6,1],[1,2],[5,4]],
            [[7,9],[2,-3],[-1,3]]
        ]
    ]
)
```

```
    ],
    #"第2个3行3列2深度的三维张量"
    [
        [[1,4],[9,3],[1,1]],
        [[1,1],[1,2],[3,3]],
        [[2,1],[3,6],[4,2]]
    ]
    ],tf.float32
)

#"2行2列的池化掩码，在行方向上的移动步长为2，在列方向上的移动步长为2"
valued3d_maxPool=tf.nn.max_pool(value3d,(1,2,2,1),[1,2,2,1],'SAME')
#"创建会话"
session=tf.Session()
#"打印结果"
print(session.run(valued3d_maxPool))
```

打印结果如下：

```
[
#"第1个三维张量池化的结果"
[
[[ 6.  5.],[ 8.  4.]],
[[ 7.  9.],[-1.  3.]]
],
#"第2个三维张量池化的结果"
[
[[ 9.  4.],[ 3.  3.]]
[[ 3.  6.],[ 4.  2.]]
]
]
```

接下来介绍另一种池化操作：平均值池化。

9.1.4 same 平均值池化

平均值池化与最大值池化类似，将计算每个位置邻域掩码内的最大值改为计算每个位置邻域掩码内的平均值即可。我们仍以 9.1.2 节中示例的张量和掩码为例，且邻域掩码沿行和

沿列的移动步长均为 2，same 平均值池化过程如图 9-8 所示。

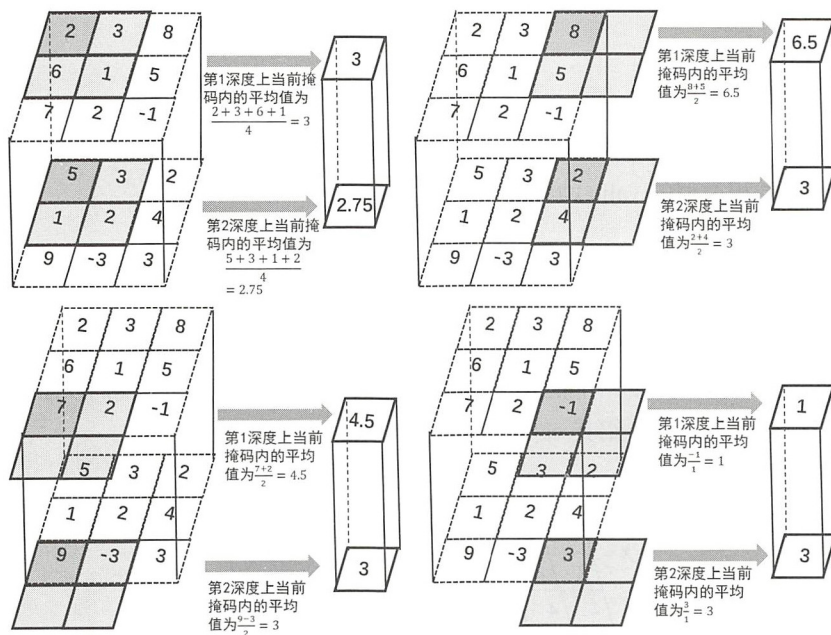


图 9-8 same 平均值池化过程

same 平均值池化的结果如图 9-9 所示。

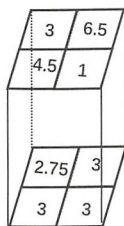


图 9-9 same 平均值池化的结果

TensorFlow 通过函数 `tf.nn.avg_pool` 实现平均值池化，使用方法和函数 `tf.nn.max_pool` 类似，以上示例的代码如下：

```
valued3d_avgPool=tf.nn.avg_pool(value3d,(1,2,2,1),[1,2,2,1],'SAME')
```

打印结果如下：

```
[[[ 3.    2.75]
```


[6.5 3.]]

[[4.5 3.]
[-1. 3.]]]]

至此，我们介绍了 same 最大值池化和平均值池化及其对应函数的使用方法。接下来，我们介绍与 valid 卷积类似的 valid 池化。

9.2 valid 池化

valid 池化与 same 池化的不同之处在于邻域掩码只在张量内移动，我们以图 9-10 所示的张量 **X** 和邻域掩码为例，介绍其 valid 的最大值池化。

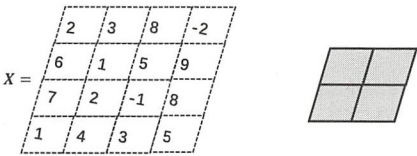


图 9-10 张量 **X** 和邻域掩码

假设邻域掩码在张量内沿行和沿列的移动步长均为 1，计算邻域掩码内的最大值，过程如图 9-11 所示。

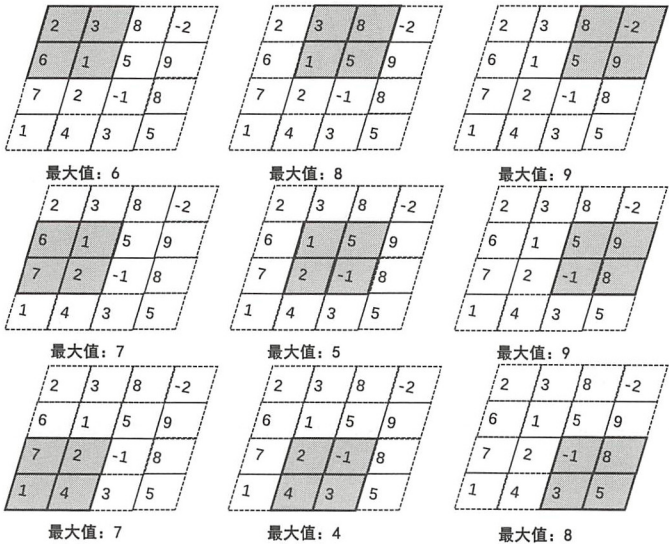


图 9-11 沿行和沿列的移动步长均为 1 的 valid 最大值池化过程

对应的代码如下：

```
valued3d_maxPool=tf.nn.max_pool(value3d,(1,2,2,1),[1,1,1,1],'VALID')
```

假设邻域掩码在张量内沿行和沿列的移动步长均为 2，则 valid 最大值池化过程如图 9-12 所示。

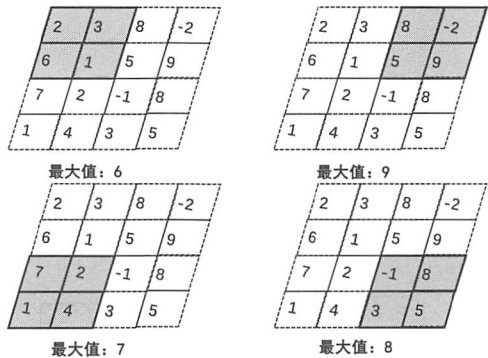


图 9-12 沿行和沿列的移动步长均为 2 的 valid 最大值池化过程

对应的代码如下：

```
valued3d_maxPool=tf.nn.max_pool(value3d,(1,2,2,1),[1,2,2,1],'VALID')
```

假设邻域掩码在张量内沿行的移动步长为 1，沿列的移动步长为 2，则 valid 最大值池化过程如图 9-13 所示。

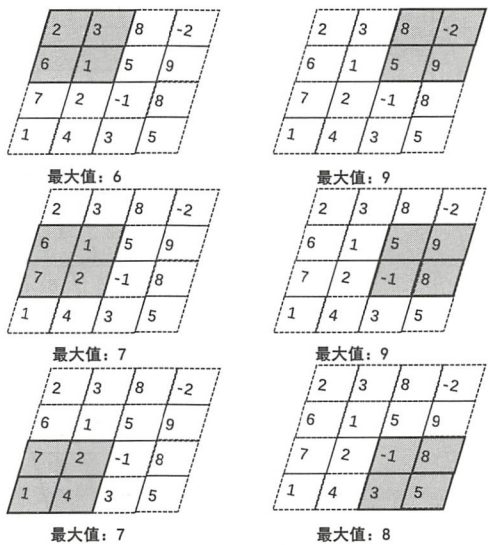


图 9-13 沿行的移动步长为 1、沿列的移动步长为 2 的 valid 最大值池化过程

对应的代码如下：

```
valued3d_maxPool=tf.nn.max_pool(value3d,(1,2,2,1),[1,1,2,1], 'VALID')
```

以上介绍的是深度为 1 的张量的 valid 池化。多深度张量的 valid 池化，本质上也是分别对每一深度进行 valid 池化操作。

9.2.1 多深度张量的 valid 池化

我们以图 9-14 所示的 3 行 3 列 2 深度的三维张量和 2 行 2 列的邻域掩码的 valid 最大值池化为例，介绍多深度的 valid 池化操作，其中邻域掩码沿行和沿列的移动步长均为 1，具体过程如图 9-15 所示。

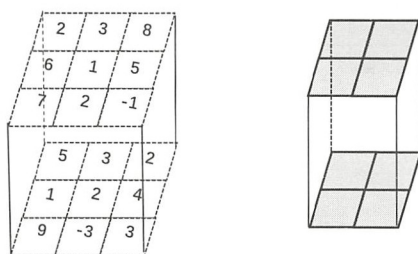


图 9-14 多深度张量和邻域掩码

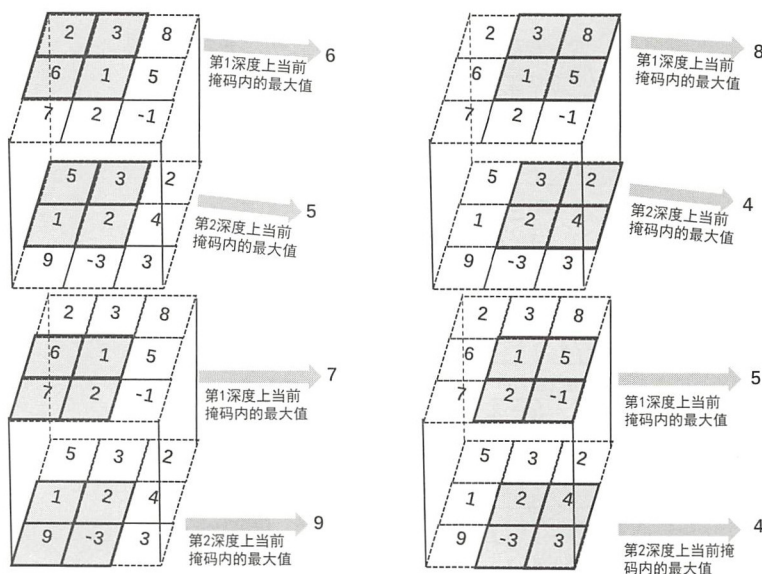


图 9-15 valid 最大值池化过程

valid 最大值池化结果如图 9-16 所示。

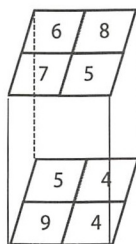


图 9-16 valid 最大值池化结果

H 行 W 列的张量与 FH 行 FW 列的邻域掩码 valid 池化, 假设邻域掩码在列方向上的移动步长为 SW , 行方向上的移动步长为 SH , 则 valid 池化结果的尺寸为: $\text{ceil}(\frac{H-(FH-1)}{SH})$ 行 $\text{ceil}(\frac{W-(FW-1)}{SW})$ 列。

9.2.2 多个三维张量的 valid 池化

最大值池化操作函数 `tf.nn.max_pool` 可以同时分别计算多个三维张量的池化操作。图 9-17 所示为两个深度为 2 的三维张量分别与 2 行 2 列邻域掩码的 same 最大值池化操作, 其沿行和沿列方向上的移动步长为 2。

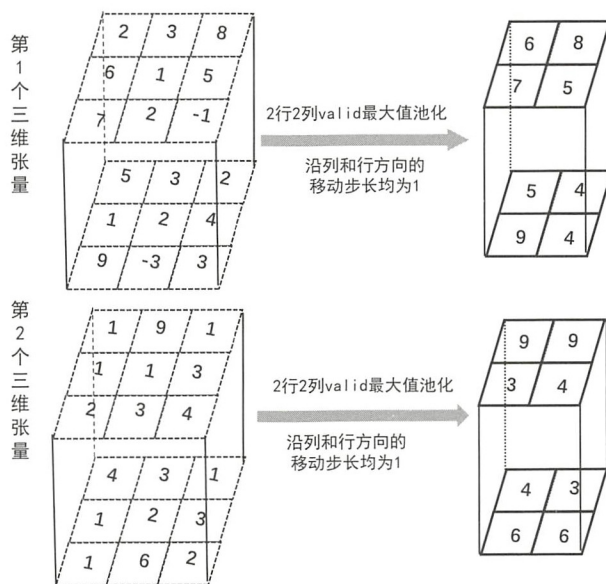


图 9-17 多个三维张量的 valid 最大值池化

只要修改 9.1.3 节中池化函数的参数即可得到以上示例对应的代码：

```
tensors_maxPool=tf.nn.max_pool(tensors,(1,2,2,1),[1,1,1,1],'VALID')
```

打印结果如下：

```
[  
#"第1个张量的valid池化结果"  
[[[6. 5.]  
   [8. 4.]]  
  
   [[7. 9.]  
    [5. 4.]]]  
#"第2个张量的valid池化结果"  
[[[9. 4.]  
   [9. 3.]]  
  
   [[3. 6.]  
    [4. 6.]]]]
```

valid 平均值池化和 same 平均值池化类似，将掩码内的平均值替换为掩码内的最大值即可，本节不再赘述。至此，我们介绍了卷积和池化操作，这两类操作是卷积神经网络的基础。

10

卷积神经网络

卷积神经网络与全连接神经网络类似，可以理解为一种变换，这种变换一般由卷积、池化、加法、激活函数等一系列操作组合而成。本章先介绍比较简单的卷积神经网络及其背后的计算过程，再介绍一些经典的卷积神经网络结构。

10.1 浅层卷积神经网络

我们可以将全连接神经网络理解为针对向量的变换，接下来我们通过第 8 章介绍的卷积操作和第 9 章介绍的池化操作的组合完成针对 3 行 3 列 2 深度的三维张量的变换，如图 10-1 所示。

针对图 10-1 所示的张量变换，我们假设输入的是如图 10-2 所示的三维张量，看看最终的变换结果是什么。

输入的三维张量首先与 3 个 2 行 2 列 2 深度的卷积核进行步长为 1 的 same 卷积，输出结果的尺寸是 3 行 3 列 3 深度，如图 10-3 所示。

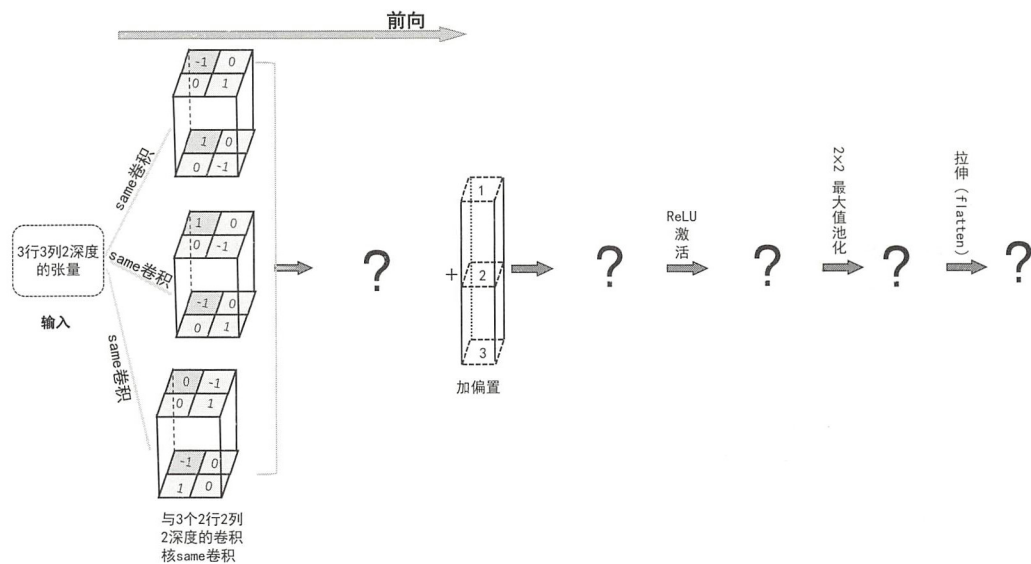


图 10-1 张量变换

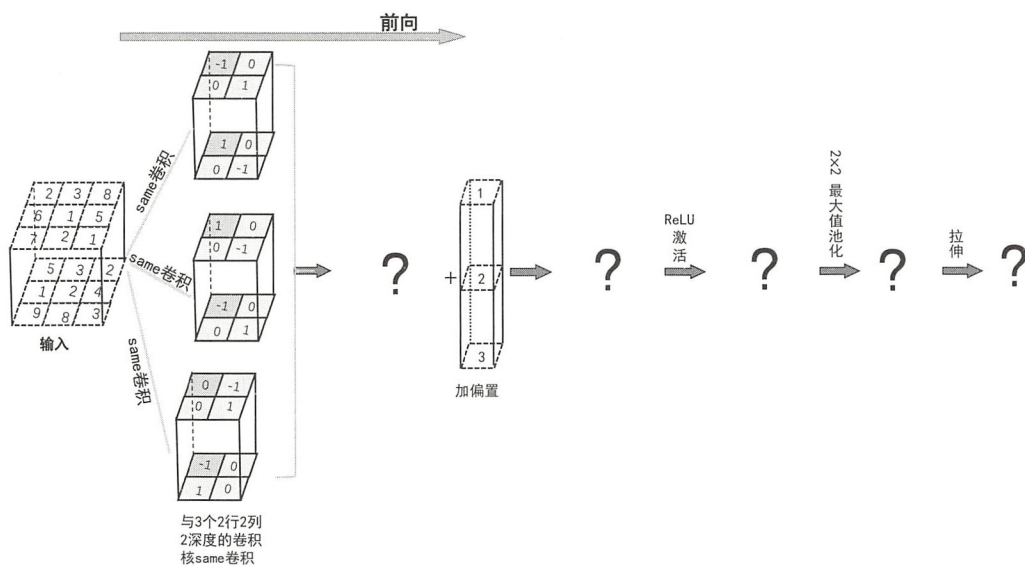


图 10-2 输入 1 个 3 行 3 列 2 深度的三维张量

对应代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
#"输入张量"
```

```

input_tensor=tf.constant([
    # "第1个高为3、宽为3、深度为2的三维张量"
    [
        [[2,5],[3,3],[8,2]],
        [[6,1],[1,2],[5,4]],
        [[7,9],[2,8],[1,3]]
    ],tf.float32)

#"3个高为2、宽为2、深度为2的卷积核"
kernel=tf.constant(
    [
        [ [-1,1,0],[1,-1,-1] ],[ [0,0,-1],[0,0,0] ] ],
        [ [ [0,0,0],[0,0,1] ], [ [1,-1,1],[-1,1,0] ] ]
    ],tf.float32
)

#"卷积"
conv2d=tf.nn.conv2d(input_tensor,kernel,(1,1,1,1),'SAME')

```

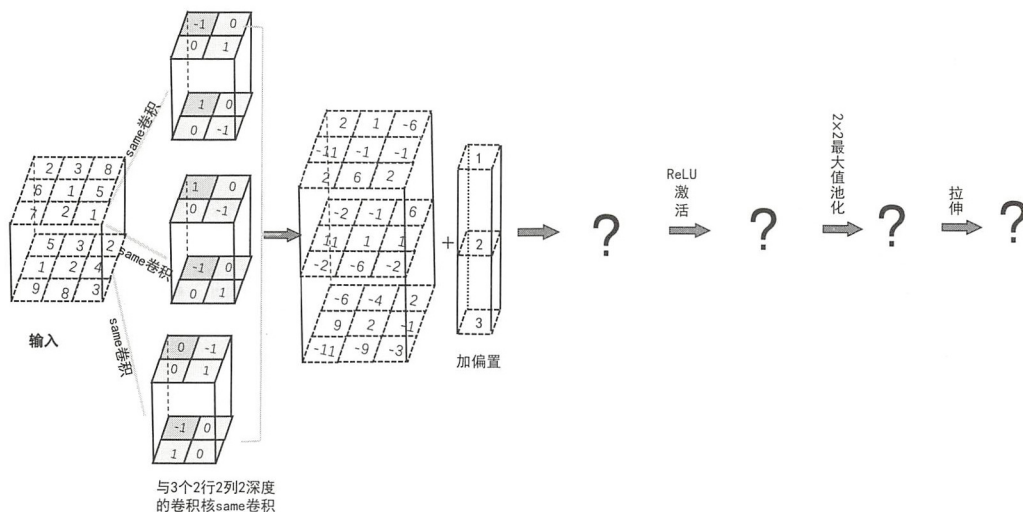


图 10-3 与3个2行2列2深度的卷积核 same 卷积

将得到的卷积结果在每一个深度上加一个常数（即加偏置），其结果如图 10-4 所示。

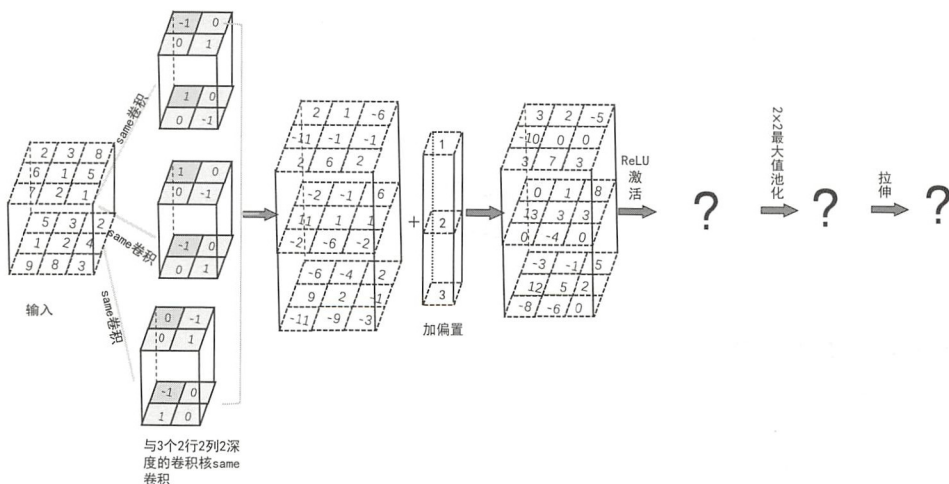


图 10-4 在每一个深度上分别加一个偏置

对应代码如下：

```
# "偏置"
bias=tf.constant([1,2,3],tf.float32)
conv2d_add_bias=tf.add(conv2d,bias)
```

用 ReLU 激活函数处理得到的结果，结果如图 10-5 所示。

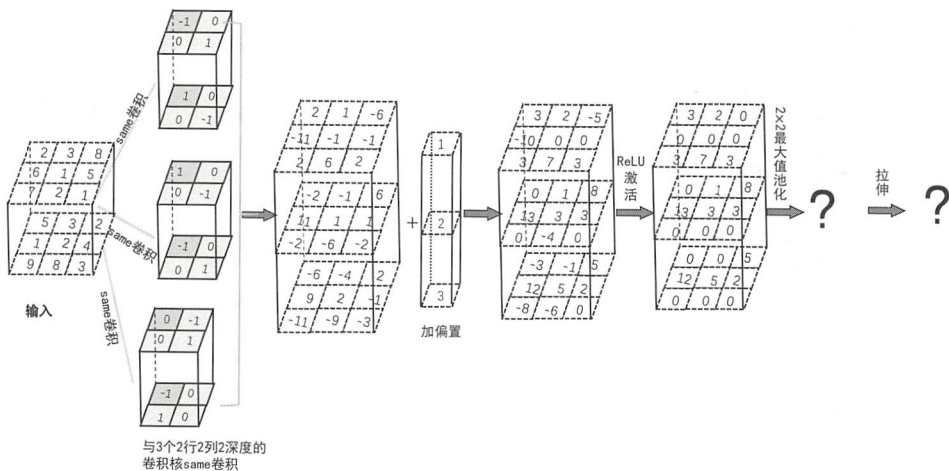


图 10-5 ReLU 激活

对应的代码如下：

```
# "激活函数"
```

```
active=tf.nn.relu(conv2d_add_bias)
```

将激活后的结果，经过 2×2 的步长为 1 的 valid 最大值池化操作，如图 10-6 所示。

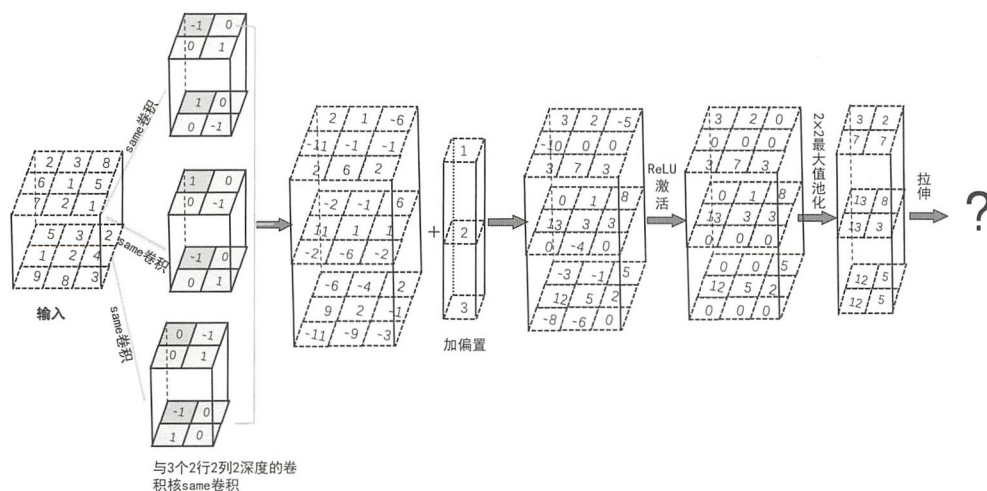


图 10-6 2×2 的 valid 最大值池化

#"pool 操作"

```
active_maxPool=tf.nn.max_pool(active,(1,2,2,1),(1,1,1,1),'VALID')
```

然后将结果进行拉伸操作，如图 10-7 所示。

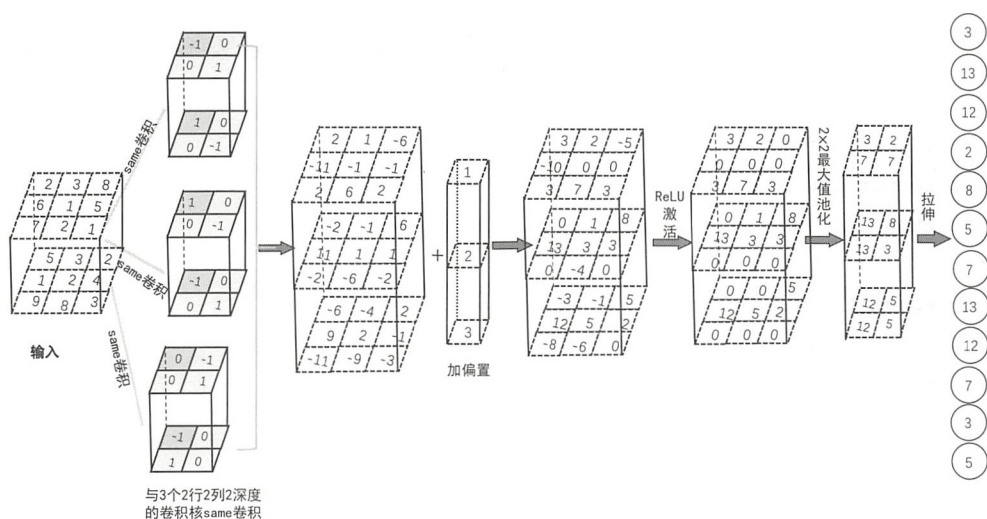


图 10-7 拉伸

对应的代码如下：

```
#"拉伸"
shape=active_maxPool.get_shape ()
num=shape[1].value*shape[2].value*shape[3].value
flatten=tf.reshape(active_maxPool,[-1,num])
#"打印结果"
session=tf.Session()
print(session.run(flatten))
```

打印结果如下：

```
[[ 3. 13. 12.  2.  8.  5.  7. 13. 12.  7.  3.  5.]]
```

至此，输入的三维张量经过图 10-1 至图 10-7 所示的一系列变换后，转换为了长度为 12 的向量，上述变换称为卷积神经网络。本节介绍的网络很简单，只有一次卷积和一次池化操作，后续介绍的经典卷积神经网络只不过是卷积和池化操作的次数多了一些，原理不变。以上示例的变换结果如图 10-8 所示。

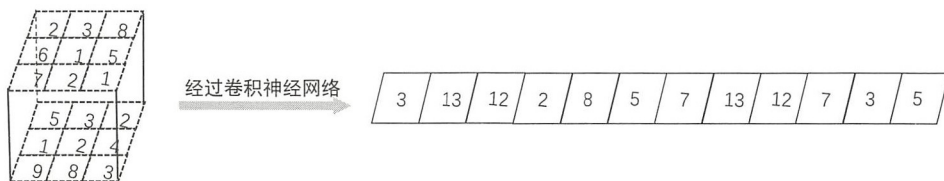


图 10-8 1 个三维张量经过卷积神经网络后的结果

上述示例中只有一个输入经过该网络结构，接下来我们介绍如何实现输入 3 个不同的三维张量，即这 3 个张量分别经过该网络的结构，具体代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
import numpy as np
#"输入张量"
input_tensor=tf.placeholder(tf.float32,[None,3,3,2])
#"3个高为2、宽为2、深度为2的卷积核"
kernel=tf.constant(
    [
        [ [-1,1,0],[1,-1,-1] ],[ [0,0,-1],[0,0,0] ] ],
        [ [ [0,0,0],[0,0,1] ], [ [1,-1,1],[-1,1,0] ] ]
    ],tf.float32
```



```

    )
    #"卷积"
    conv2d=tf.nn.conv2d(input_tensor,kernel,(1,1,1,1),'SAME')
    #"偏置"
    bias=tf.constant([1,2,3],tf.float32)
    conv2d_add_bias=tf.add(conv2d,bias)
    #"激活函数"
    active=tf.nn.relu(conv2d_add_bias)
    #"pool操作"
    active_maxPool=tf.nn.max_pool(active,(1,2,2,1),(1,1,1,1),'VALID')
    #"拉伸"
    shape=active_maxPool.get_shape()
    num=shape[1].value*shape[2].value*shape[3].value
    flatten=tf.reshape(active_maxPool,[-1,num])
    #flatten=tf.contrib.layers.flatten(active_maxPool)
    #"打印结果"
    session=tf.Session()
    print(session.run(flatten,feed_dict={
        input_tensor:np.array([
            #"第1个3行3列2深度的三维张量"
            [
                [[2,5],[3,3],[8,2]],
                [[6,1],[1,2],[5,4]],
                [[7,9],[2,8],[1,3]]
            ],
            #"第2个3行3列2深度的三维张量"
            [
                [[1,2],[3,6],[1,2]],
                [[3,1],[1,2],[2,1]],
                [[4,5],[2,7],[1,2]]
            ],
            #"第3个3行3列2深度的三维张量"
            [
                [[2,3],[3,2],[1,2]],
                [[4,1],[3,2],[1,2]],
                [[1,0],[4,1],[4,3]]
            ]
        ])
    })

```




```
],
],np.float32)))))
```

打印结果如下：

```
[[ 3.  13.  12.   2.   8.   5.   7.  13.  12.   7.   3.   5.]
 [ 5.   9.   8.   5.   3.   7.   6.   9.   8.   6.   3.   7.]
 [ 3.   4.   5.   2.   4.   5.   1.   5.   5.   2.   5.   5.]]
```

这 3 个三维张量分别得到了相对应的输出，在 TensorFlow 实现卷积神经网络的代码中，这 3 个三维张量就可以看成 1 个四维张量，输出结果是 1 个 3 行 12 列的值，如图 10-9 所示，右图中的每一行代表每一个三维张量经过卷积神经网络后的输出结果。

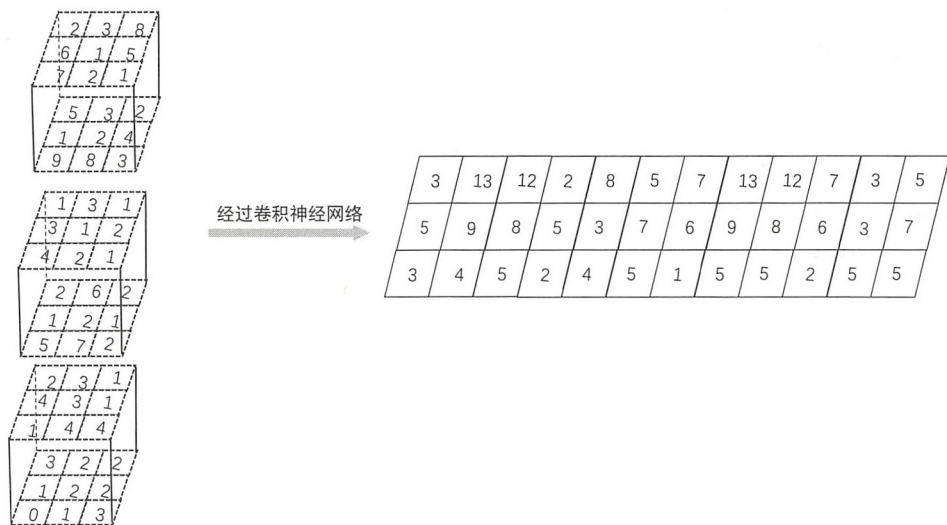


图 10-9 3 个三维张量分别经过卷积神经网络后的结果

了解了简单的卷积神经网络后，接下来介绍经典的卷积神经网络结构。

10.2 LeNet

LeNet^[1] 是第一个成熟的卷积神经网络，是专门为处理 MNIST 数字字符集的分类问题而设计的网络结构，该网络结构输入值的尺寸是 32 行 32 列 1 深度的三维张量（灰度图像），其经过 LeNet 网络的变换过程如下。



第1步：与6个高为5、宽为5、深度为1的卷积核 valid 卷积，然后将卷积结果的每一深度加偏置，最后将结果经过激活函数处理，得到的结果的尺寸高为28、宽为28、深度为6，如图10-10所示。

第1步对应的代码如下，其中卷积核和偏置都由满足高斯分布的随机数生成。

```
#"6个高为5、宽为5、深度为1的卷积核"
k1=tf.Variable(tf.random_normal([5,5,1,6]),dtype=tf.float32)
c1=tf.nn.conv2d(x,k1,[1,1,1,1],'VALID')
#"长度为6（因为卷积结果的输出深度为6）的偏置"
b1=tf.Variable(tf.random_normal([6]),dtype=tf.float32)
#"c1与b1求和，并输入激活函数"
c1_b1=tf.add(c1,b1)
r1=tf.nn.relu(c1_b1)
```

第2步：进行 2×2 的步长为2的 valid 最大值池化，池化后结果的尺寸高为14、宽为14、深度为6，如图10-11所示。

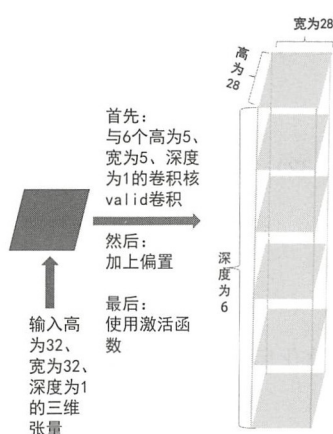


图 10-10 LeNet 网络结构计算的第1步

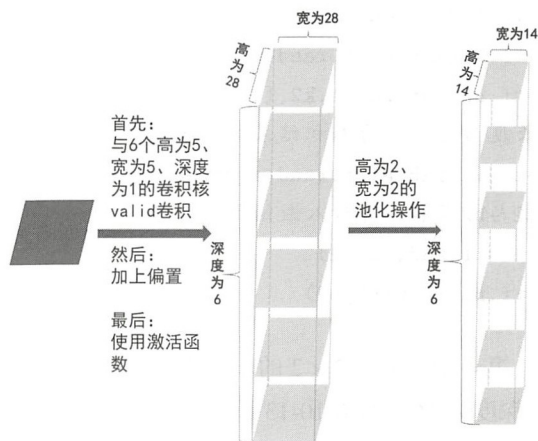


图 10-11 LeNet 网络结构计算的第2步

第2步的对应代码如下：

```
#"最大值池化操作，池化掩码的尺寸为高2宽2，步长为2"
p1=tf.nn.max_pool(r1,[1,2,2,1],[1,2,2,1],'VALID')
```

第3步：先与16个高为5、宽为5、深度为6的卷积核 valid 卷积，然后将卷积结果在每一深度上加偏置，最后将加上偏置的结果经过激活处理。得到的结果的尺寸高为10、宽为10、深度为16，如图10-12所示。



图解深度学习与神经网络：从张量到 TensorFlow 实现

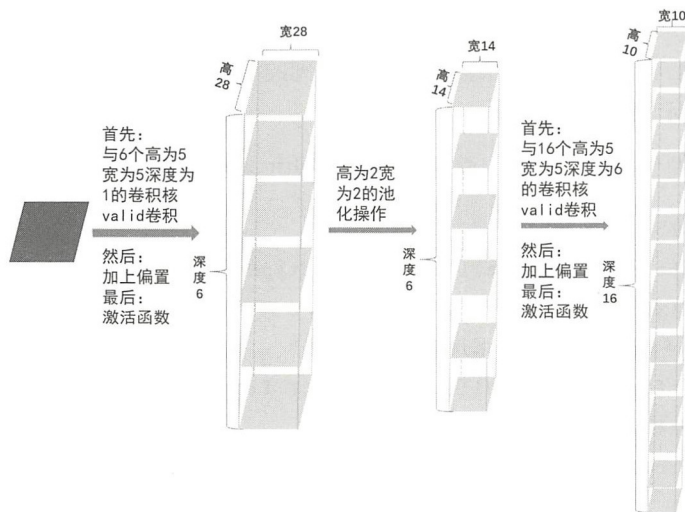


图 10-12 LeNet 网络结构计算的第 3 步

第 3 步对应的代码如下，其中卷积核和偏置都是由满足高斯分布的随机数生成的：

```
#"p1与16个高为5、宽为5、深度为6的卷积核的卷积"
k2=tf.Variable(tf.random_normal([5,5,6,16]),dtype=tf.float32)
c2=tf.nn.conv2d(p1,k2,[1,1,1,1],'VALID')
#"长度为16（因为卷积结果的输出深度为16）的偏置"
b2=tf.Variable(tf.random_normal([16]),dtype=tf.float32)
#"c2与b2求和，并输入激活函数"
c2_b2=tf.add(c2,b2)
r2=tf.nn.relu(c2_b2)
```

第 4 步：进行 2×2 的步长为 2 的 valid 最大值池化，池化后结果的尺寸是高为 5、宽为 5、深度为 16，如图 10-13 所示。

第 4 步对应的代码如下：

```
#"最大值池化操作，池化掩码的尺寸是高为2、宽为2，步长为2"
p2=tf.nn.maxpool(c2,[1,2,2,1],[1,2,2,1],'VALID')
```

第 5 步：将第 4 步得到的结果拉伸为 1 个一维向量，其长度为 $5 \times 5 \times 16 = 400$ ，然后将这个向量经过一个全连接神经网络处理，该全连接神经网络有 2 个隐含层，其中输入层有 400 个神经元，第 1 个隐含层有 120 个神经元，第 2 个隐含层有 10 个神经元。因为要处理的数据有 10 个类别，所以输出层有 10 个神经元，如图 10-14 所示。



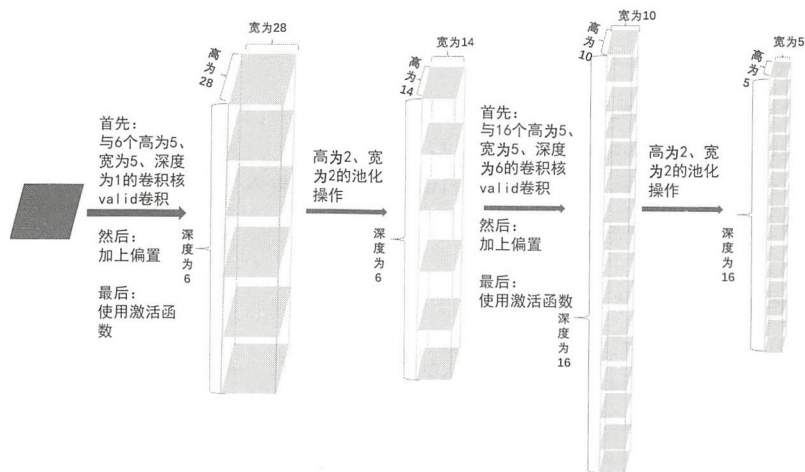


图 10-13 LeNet 网络结构计算的第 4 步

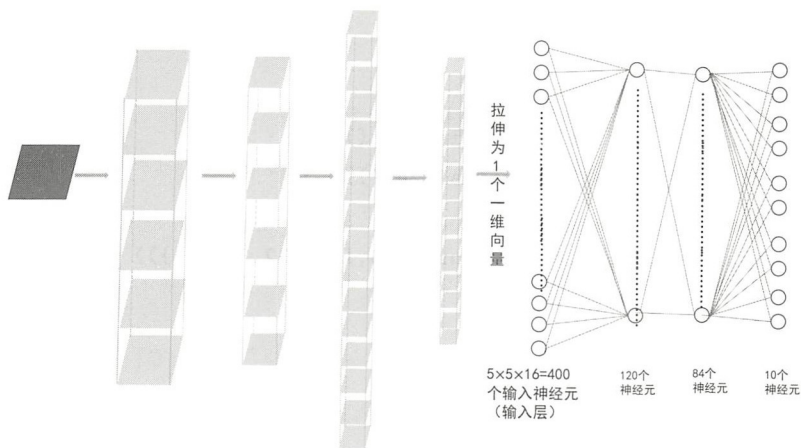


图 10-14 LeNet 网络结构

经过这 5 步变换，我们将 1 个高为 32、宽为 32、深度为 1 的张量转换成了 1 个长度为 10 的向量，其中第 5 步对应的代码如下：

```
#"拉伸为一维张量，作为一个全连接神经网络的输入"
flatten_p2=tf.reshape(p2,[-1,5*5*16])
#"第1层的权重矩阵和偏置"
w1=tf.Variable(tf.random_normal([5*5*16,120]))
bw1=tf.Variable(tf.random_normal([120]))
#"第1层的线性组合与偏置求和，并作为relu激活函数的输入"
h1=tf.add(tf.matmul(flatten_p2,w1),bw1)
```



图解深度学习与神经网络：从张量到 TensorFlow 实现

```

sigma1=tf.nn.relu(h1)
#"第2层的权重矩阵和偏置"
w2=tf.Variable(tf.random_normal([120,84]))
bw2=tf.Variable(tf.random_normal([84]))
#"第2层的线性组合与偏置求和，并作为relu激活函数的输入"
h2=tf.add(tf.matmul(sigma1,w2),bw2)
sigma2=tf.nn.relu(h2)
#"第3层的线性组合与偏置求和"
w3=tf.Variable(tf.random_normal([84,10]))
bw3=tf.Variable(tf.random_normal([10]))
h3=tf.add(tf.matmul(sigma2,w3),bw3)
#"将h3作为sigmoid激活函数的输入，作为最后输出层的输出"
out=tf.nn.sigmoid(h3)

#"创建会话"
session=tf.Session()
#"初始化变量"
session.run(tf.global_variables_initializer())
session.run(tf.local_variables_initializer())
#"打印结果如下："
print(session.run(out,{x:np.random.normal(0,1,[2,32,32,1])}))

```

打印结果如下：

```

[[0. 1. 1. 1. 1. 0. 1. 1. 1. 1.]
 [0. 1. 0. 1. 1. 0. 1. 1. 1. 0.]]

```

上述代码中，随机生成了 2 个宽为 32、高为 32、深度为 1 的 ndarray 作为输入图像，打印结果为 1 个 2 行 10 列的二维张量，即每一行代表经过 LeNet 网络变换后的最终输出结果。

同第 6 章介绍的全连接神经网络解决分类问题类似，针对利用 LeNet 网络解决分类问题，我们希望找到适合的卷积核、权重矩阵和偏置，使得任意一张 MNIST 数字图像经过该网络变换后，都能得到相应的图片标签，如图 10-15 所示。或者用常用的 softmax 处理，将 LeNet 网络的输出值经过 softmax 处理，得到相应的图片标签，但是完全满足该条件的卷积核、权重矩阵和偏置几乎不存在，所以同样需要构造损失函数，即把卷积核、权重矩阵、偏置中的每一个值看作损失函数的参数，然后利用各种梯度下降法求解这些值，这一部分的思路同第 6 章介绍的全连接神经网络解决分类问题相同，连代码都相同，所以不再赘述。构造损失函数后，如果每一次迭代都需要针对每一个参数分别进行梯度下降，那么复杂度高且计



算时间长。幸运的是，与全连接神经网络类似，卷积神经网络也有相应地根据导数链式法则推导出的 BP 算法，可以快速地计算出每一个参数的梯度，该算法的核心内容将在第 11 章和第 12 章详细介绍。

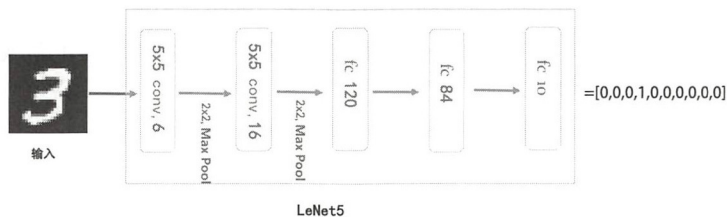


图 10-15 利用 LetNet 网络结构进行分类

为了方便介绍，我们用图 10-16 表示 LetNet 网络的结构。

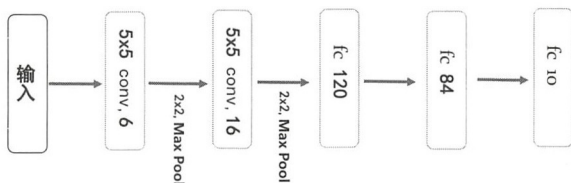


图 10-16 LetNet 网络的结构

图 10-17 所示为图 10-16 中的一部分，常称为**卷积层**，卷积层一般会包含 3 个步骤。

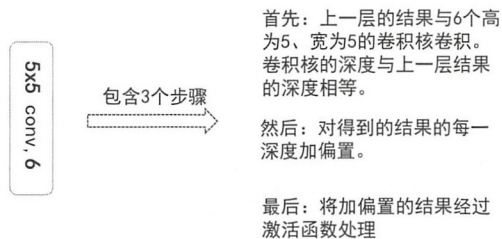


图 10-17 卷积层

图 10-18 所示为全连接层的示意图。

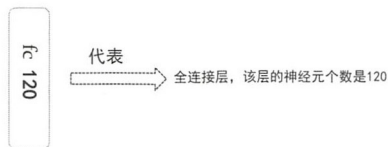


图 10-18 全连接层

因为 LeNet 网络结构主要包括这五层（卷积层和全连接层），所以也常称为 LeNet5。



至此，我们已经多次使用了 ReLU 激活函数，但是第一次将 ReLU 激活函数运用到卷积网络是 10.3 节将介绍的 AlexNet^[2]，该网络结构的提出也是卷积网络里程碑式的突破。从 1989 年的 LetNet 到 2012 年 AlexNet 的提出，在这中间的二十多年，其他机器学习方法，如支持向量机（SVM）的表现超过了神经网络和卷积神经网络，但是随后大数据、硬件、算法（如 ReLU 激活函数的提出）这三大壁垒的突破，使得卷积神经网络又重新焕发了生机，拉开了深度学习技术发展的序幕。

10.3 AlexNet

AlexNet 卷积神经网络主要是针对高为 224、宽为 224 的彩色图像的分类问题设计的卷积神经网络结构。AlexNet 比 LeNet 具备更深的网络结构，卷积核的深度也更深，全连接层的神经元个数也更多，接下来详细介绍该网络结构。

10.3.1 AlexNet 网络结构详解

AlexNet 网络结构如图 10-19 所示，我们可以将其理解为针对 224 行 224 列 3 深度的三维张量的变换。



图 10-19 AlexNet 对三维张量的变换过程

AlexNet 卷积神经网络处理的是 1000 个分类的任务，所以最后全连接神经网络的输出层的神经元个数是 1000，只要根据图 10-19 所示的具体步骤操作，即可完成相应的代码，具体过程如下：



```

# -*- coding: utf-8 -*-
import tensorflow as tf
import numpy as np
#"输入"
x=tf.placeholder(tf.float32,[None,224,224,3])
keep_prob=tf.placeholder(tf.float32)
#"第1步：与96个11×11×3的卷积核卷积。第2步：加上偏置。第3步：使用激活函数"
w1=tf.Variable(tf.random_normal([11,11,3,96]),dtype=tf.float32,name='w1')
l1=tf.nn.conv2d(x,w1,[1,4,4,1],'SAME')
b1=tf.Variable(tf.random_normal([96]),dtype=tf.float32,name='b1')
l1=tf.nn.bias_add(l1,b1)
l1=tf.nn.relu(l1)
#"2×2最大值池化操作，移动步长为2"
pool_l1=tf.nn.max_pool(l1,[1,2,2,1],[1,2,2,1],'SAME')
#"第1步：与256个5×5×96的卷积核卷积。第2步：加上偏置。第3步：使用激活函数"
w2=tf.Variable(tf.random_normal([5,5,96,256]),dtype=tf.float32,name='w2')
l2=tf.nn.conv2d(pool_l1,w2,[1,1,1,1],'SAME')
b2=tf.Variable(tf.random_normal([256]),dtype=tf.float32,name='b2')
l2=tf.nn.bias_add(l2,b2)
l2=tf.nn.relu(l2)
#"2×2最大值池化操作，移动步长为2"
pool_l2=tf.nn.max_pool(l2,[1,2,2,1],[1,2,2,1],'SAME')
#"第1步：与384个3×3×256的卷积核卷积。第2步：加上偏置。第3步：使用激活函数"
w3=tf.Variable(tf.random_normal([3,3,256,384]),dtype=tf.float32,name='w3')
l3=tf.nn.conv2d(pool_l2,w3,[1,1,1,1],'SAME')
b3=tf.Variable(tf.random_normal([384]),dtype=tf.float32,name='b3')
l3=tf.nn.bias_add(l3,b3)
l3=tf.nn.relu(l3)
#"第1步：与384个3×3×384的卷积核卷积。第2步：加上偏置。第3步：使用激活函数"
w4=tf.Variable(tf.random_normal([3,3,384,384]),dtype=tf.float32,name='w4')
l4=tf.nn.conv2d(l3,w4,[1,1,1,1],'SAME')
b4=tf.Variable(tf.random_normal([384]),dtype=tf.float32,name='b4')
l4=tf.nn.bias_add(l4,b4)
l4=tf.nn.relu(l4)
#"第1步：与256个3×3×384的卷积核卷积。第2步：加上偏置。第3步：使用激活函数"
w5=tf.Variable(tf.random_normal([3,3,384,256]),dtype=tf.float32,name='w5')

```



图解深度学习与神经网络：从张量到 TensorFlow 实现

```

l5=tf.nn.conv2d(l4,w5,[1,1,1,1],'SAME')
b5=tf.Variable(tf.random_normal([256]),dtype=tf.float32,name='b5')
l5=tf.nn.bias_add(l5,b5)
l5=tf.nn.relu(l5)
#"2x2最大值池化操作，移动步长为2"
pool_l5=tf.nn.max_pool(l5,[1,2,2,1],[1,2,2,1],'SAME')
#"拉伸，作为全连接神经网络的输入层"
pool_l5_shape=pool_l5.get_shape()
num=pool_l5_shape[1].value*pool_l5_shape[2].value*pool_l5_shape[3].value
flatten=tf.reshape(pool_l5,[-1,num])
#"第1个隐含层"
fcW1=tf.Variable(tf.random_normal([num,4096]),dtype=tf.float32,name='fcW1')
fc_l1=tf.matmul(flatten,fcW1)
fcb1=tf.Variable(tf.random_normal([4096]),dtype=tf.float32,name='fcb1')
fc_l1=tf.nn.bias_add(fc_l1,fcb1)
fc_l1=tf.nn.relu(fc_l1)
fc_l1=tf.nn.dropout(fc_l1,keep_prob)
#"第2个隐含层"
fcW2=tf.Variable(tf.random_normal([4096,4096]),dtype=tf.float32,name='fcW2')
fc_l2=tf.matmul(fc_l1,fcW2)
fcb2=tf.Variable(tf.random_normal([4096]),dtype=tf.float32,name='fcb2')
fc_l2=tf.nn.bias_add(fc_l2,fcb2)
fc_l2=tf.nn.relu(fc_l2)
fc_l2=tf.nn.dropout(fc_l2,keep_prob)
#"输出层"
fcW3=tf.Variable(tf.random_normal([4096,1000]),dtype=tf.float32,name='fcW3')
out=tf.matmul(fc_l2,fcW3)
fcb3=tf.Variable(tf.random_normal([1000]),dtype=tf.float32,name='fcb3')
out=tf.nn.bias_add(out,fcb3)
out=tf.nn.relu(out)
#"创建会话"
session=tf.Session()
session.run(tf.global_variables_initializer())
result=session.run(out,feed_dict={x:np.ones([2,224,224,3],np.float32),
                                   keep_prob:0.5})
#"打印最后的输出尺寸"

```

```
print(np.shape(result))
```

打印结果如下：

```
(2, 1000)
```

AlexNet 一共包括 5 个卷积层和 3 个全连接层，所以，介绍 AlexNet 时，一般会说该网络有 8 层，如图 10-20 所示。

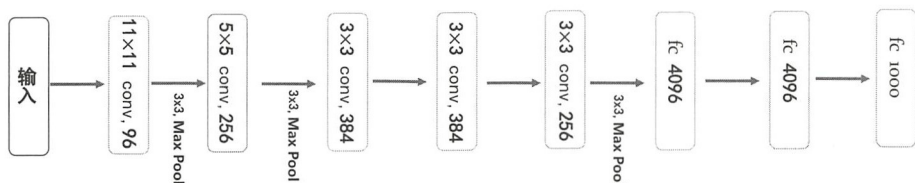


图 10-20 AlexNet 卷积神经网络的结构

AlexNet 在算法上有两个非常重要的改进，直到现在还被广泛应用。第一个改进是提出 ReLU 激活函数，该激活函数的优点已经在第 6 章详细介绍过，本节不再赘述。第二个改进是，提出 dropout，该操作可以有效地防止网络的过拟合，所谓的过拟合就是网络结构在训练集上的正确率很高，但在测试集上的准确率较低。接下来，我们详细介绍 dropout 是怎样的操作，以及对应的 TensorFlow 实现。

10.3.2 dropout 及其梯度下降

1. dropout 函数的定义

我们先介绍 TensorFlow 中的函数：

```
dropout(x, keep_prob, noise_shape=None, seed=None, name=None)
```

其中参数 x 代表输入张量，参数 $keep_prob$ 属于区间 $(0, 1]$ 中的一个值，输入张量 x 中每一个值以概率 $keep_prob$ 变为原来的 $1/keep_prob$ 倍，以概率 $1 - keep_prob$ 变为 0。

我们以图 10-21 所示的 2 行 4 列的二维张量为例，假设该张量作为函数 `dropout` 的参数 x 的值，令参数 $keep_prob=0.5$ ，对应的代码如后所示。

1	3	2	6
7	5	4	9

图 10-21 2 行 4 列的二维张量

图解深度学习与神经网络：从张量到 TensorFlow 实现

```
# -*- coding: utf-8 -*-
import tensorflow as tf
#"输入的二维张量"
t=tf.constant(
    [
        [1,3,2,6],
        [7,5,4,9]
    ],tf.float32
)
#"dropout处理"
r=tf.nn.dropout(t,0.5)
#"创建会话"
session=tf.Session()
#"打印结果"
print(session.run(r))
```

打印结果如下：

```
[[ 0.  6.  0. 12.]
 [14.  0.  0. 18.]]
```

上述打印结果如图 10-22 所示。

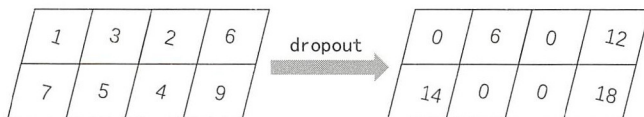


图 10-22 2 行 4 列的二维张量的 dropout 处理

每次运行的结果可能有所不同。显然，张量中的值要么变为原来的 $\frac{1}{0.5}$ 倍，要么变为 0。

在以上示例的基础上，我们来介绍参数 `noise_shape` 的作用，因为张量的尺寸为 2 行 4 列，如果令参数 `noise_shape=[2,1]`，则代表把每一行看成一个整体，同一行的值要么全变为原来的 $1/\text{keep_prob}$ 倍，要么全变为 0，代码如下：

```
r=tf.nn.dropout(t,0.5,noise_shape=[2,1])
```

打印结果如下：

```
[[ 0.  0.  0.  0.]
 [14. 10.  8. 18.]]
```


每次运行的输出结果可能有所不同。

同理，如果参数 `noise_shape=[1,4]`，代表把每一列看成一个整体，同一列的值要么全变为原来的 $1/\text{keep_prob}$ 倍，要么全变为 0，代码如下：

```
r=tf.nn.dropout(t,0.5,noise_shape=[1,4])
```

打印结果如下：

```
[[ 2.  0.  4.  0.]
 [14.  0.  8.  0.]]
```

每次运行输出结果可能所有不同。接下来介绍 dropout 在卷积神经网络中的应用。

2. dropout 层

dropout 处理一般用在全连接神经网络的全连接层或者卷积神经网络后面的全连接层。假设有如图 10-23 所示的全连接神经网络，其中激活函数为 $f(x) = x$ ，每一层的偏置均为 0。

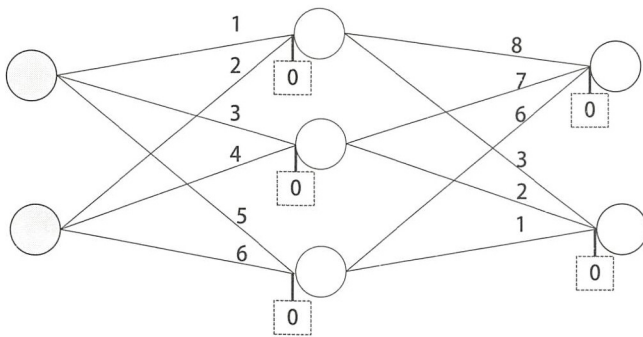


图 10-23 全连接神经网络

假设该网络有两个输入 $[2, 3]$ 和 $[1, 4]$ ，根据全连接神经网络的计算步骤，计算每一个输入在隐含层和输出层的值，如图 10-24 所示。

$$\begin{array}{ccc}
 \begin{bmatrix} 2 & 3 \\ 1 & 4 \end{bmatrix} & \xrightarrow{\quad} & \begin{bmatrix} 8 & 18 & 28 \\ 9 & 19 & 29 \end{bmatrix} & \xrightarrow{\quad} & \begin{bmatrix} 358 & 88 \\ 379 & 94 \end{bmatrix} \\
 \text{输入层} & & \text{隐含层} & & \text{输出层}
 \end{array}$$

图 10-24 计算每一个输入在隐含层和输出层的值

两个输入分别经过全连接神经网络，如图 10-25 所示。



图 10-25 两个输入分别经过全连接神经网络的变换

接着，修改图 10-23 所示的全连接神经网络，针对隐含层做 dropout 处理，如图 10-26 所示。隐含层的每一个神经元的值，以概率 p 变为原来的 $\frac{1}{p}$ 倍，其中 $p \in (0, 1]$ ，以概率 $1 - p$ 变为 0。显然，如果 $p = 1$ ，就是全连接神经网络。

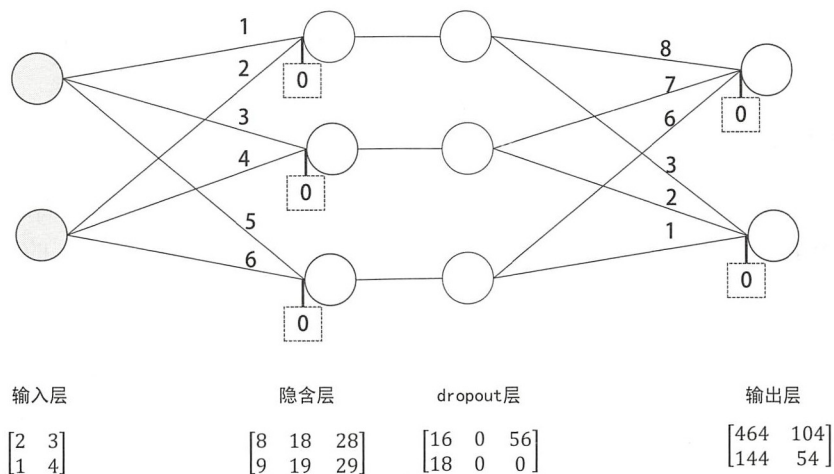


图 10-26 增加了 dropout 处理的全连接神经网络

仍以 $[2, 3]$ 和 $[1, 4]$ 作为网络的两个输入，通过以下代码介绍分别经过该网络变换后的输出值，具体如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
import numpy as np
#"占位符"
x=tf.placeholder(tf.float32,[None,2])
keep_prob=tf.placeholder(tf.float32)
#"输入层到隐含层的权重矩阵"
w1=tf.constant([
    [1,3,5],
    [2,4,6]
```

```

        ],tf.float32)
#"隐含层的值"
h1=tf.matmul(x,w1)
#"dropout层"
h1_dropout=tf.nn.dropout(h1,keep_prob)
#"dropout层到输出层的权重矩阵"
w2=tf.constant(
    [
        [8,3],
        [7,2],
        [6,1]
    ],tf.float32
)
#"输出层的值"
o=tf.matmul(h1_dropout,w2)
x_input=np.array([[2,3],[1,4]],np.float32)
#"创建会话"
session=tf.Session()
h1_arr,h1_dropout_arr,o_arr=s=session.run(
    [h1,h1_dropout,o],feed_dict={x:x_input,keep_prob:0.5})
#"打印结果"
print('隐含层的值:')
print(h1_arr)
print("'dropout层的值:'")
print(h1_dropout_arr)
print('输出层的值:')
print(o_arr)

```

打印结果如下:

```

"隐含层的值:"
[[ 8. 18. 28.]
 [ 9. 19. 29.]]
"dropout层的值:"
[[ 0. 36.  0.]
 [ 0.  0. 58.]]
"输出层的值:"

```

```
[[252.  72.]  
 [348.  58.]]
```

打印结果如图 10-27 所示。

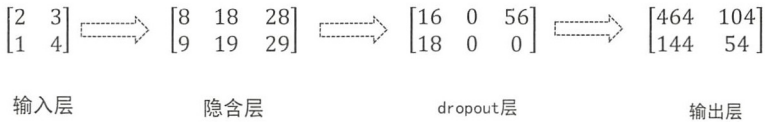


图 10-27 计算每一个输入在隐含层、dropout 层及输出层的值

当 $[2, 3]$ 作为输入层的值时，隐含层的值为 $[8, 18, 28]$ ，经过 dropout 层后，隐含层第 1 个神经元的值变为原来的 $\frac{1}{0.5}$ 倍，即 $8 \times 2 = 16$ ；隐含层第 2 个神经元的值变为 0，隐含层第 3 个神经元的值变为原来的 $\frac{1}{0.5}$ 倍，即 $28 \times 2 = 56$ 。同理，当 $[1, 4]$ 作为输出层的值时，隐含层的值为 $[9, 19, 29]$ ，经过 dropout 层后，隐含层的第 2 个和第 3 个神经元变为 0，第 1 个神经元的值变为 $9 \times 2 = 18$ ，因为以概率 p 随机变化，所以同样的输入值，每次经过该网络后，可能出现不同的输出值。每次运行以上代码时，除了“隐含层的值”不变，“dropout 层的值”和“输出层的值”可能出现不同的打印结果。显然，如果将上述代码中的 `keep_prob` 的值设置为 1，即：

```
h1_arr,h1_dropout_arr,o_arr=s=session.run(  
    [h1,h1_dropout,o],feed_dict={x:x_input,keep_prob:1})
```

打印结果和不加 dropout 层的全连接神经网络的输出结果相同。

接下来，我们详细介绍添加了 dropout 处理的全连接神经网络在训练网络时是如何进行梯度下降的。

3. dropout 梯度下降

我们通过图 10-28 所示的简单的网络结构理解有 dropout 层的全连接神经网络的梯度反向传播，其中 dropout 对应的概率 $p = 0.5$ ，输入层、隐含层、输出层都只有一个神经元，激活函数为 $\sigma(x) = x$ 。

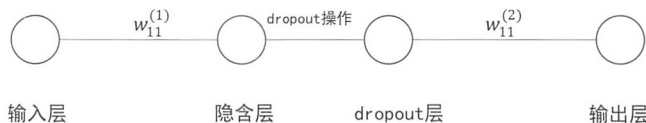


图 10-28 带有 dropout 层的全连接神经网络

根据该网络构造一个函数 f ，比如 n 个输入经过该网络后对应 n 个输出，把这 n 个输出之和作为我们构造的函数，这里假设只有一个输入，比如为 3，则可构造函数

$$f = w_{11}^{(2)} \times \text{dropout}(3 \times w_{11}^{(1)})$$

当 $\text{dropout}(3 \times w_{11}^{(1)}) = 6w_{11}^{(1)}$ 时（如图 10-29 所示）， $f = 6w_{11}^{(1)}w_{12}^{(1)}$ 。

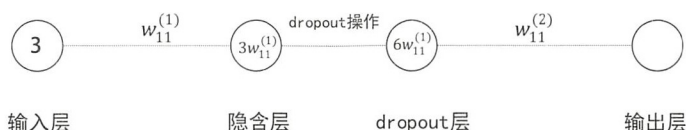


图 10-29 dropout 层的值为隐含层的 2 倍

当 $\text{dropout}(3 \times w_{11}^{(1)}) = 0$ 时（如图 10-30 所示）， $f = 0$ 。

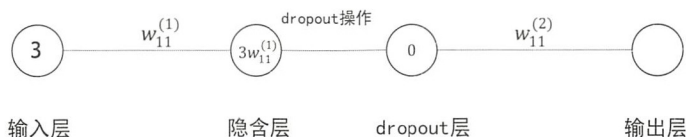


图 10-30 dropout 层的值为 0

接下来，我们初始化 $w_{11}^{(1)} = 10$ ， $w_{11}^{(2)} = 6$ ，针对该函数进行标准梯度下降，其中学习率 $\eta = 0.01$ ，具体代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
import numpy as np
#"占位符，已知数据"
x=tf.placeholder(tf.float32,(None,1))
keep_pro=tf.placeholder(tf.float32)
#"输入层到隐含层的权重矩阵"
w1=tf.Variable(tf.constant([[10]],tf.float32),dtype=tf.float32)
l1=tf.matmul(x,w1)
#"dropout层"
l1_dropout=tf.nn.dropout(l1,keep_pro)
#"隐含层到输出层的权重矩阵"
w2=tf.Variable(tf.constant([[6]],tf.float32),dtype=tf.float32)
l=tf.matmul(l1_dropout,w2)
#"利用网络输出值构造函数f"
f=tf.reduce_sum(l)
```

```
#"梯度下降法"
opti=tf.train.GradientDescentOptimizer(0.01).minimize(f)
#"网络的输入值"
x_array=np.array([[3]],np.float32)
#"4次迭代，打印每一次迭代隐含层的值和网络权重"
with tf.Session() as session:
    session.run(tf.global_variables_initializer())
    for i in range(4):
        _,l1_dropout_array=session.run([opti,l1_dropout],
                                         {x:x_array,keep_pro:0.5})
        print('----第"{}"次迭代"----'.format(i+1))
        print("'dropout层的值'")
        print(l1_dropout_array)
        print('网络当前的权重')
        print(session.run([w1,w2]))
```

打印结果如下：

```
"----第1次迭代----"
"dropout层的值"
[[60.]]
"当前网络的权重"
[array([[9.64]], dtype=float32), array([[5.4]], dtype=float32)]
"----第2次迭代----"
"dropout层的值"
[[0.]]
"当前网络的权重"
[array([[9.64]], dtype=float32), array([[5.4]], dtype=float32)]
"----第3次迭代----"
"dropout层的值"
[[57.840004]]
"当前网络的权重"
[array([[9.316]], dtype=float32), array([[4.8216]], dtype=float32)]
"----第4次迭代----"
"dropout层的值"
[[55.896]]
"当前网络的权重"
```

```
[array([[9.026704]], dtype=float32), array([[4.26264]], dtype=float32)]
```

每次运行的结果可能有所不同。接下来我们详细分析打印结果。

第 1 次迭代：从打印结果可以看出，drouput 层的值不等于 0，而是等于隐含层神经元值的 2 倍，此时函数 $f = 6w_{11}^{(1)}w_{11}^{(2)}$ ，则 f 在 $w_{11}^{(1)} = 10$ 和 $w_{11}^{(2)} = 6$ 的梯度为

$$\frac{\partial f}{\partial w_{11}^{(1)} | w_{11}^{(1)}=10} = 36, \frac{\partial f}{\partial w_{11}^{(2)} | w_{11}^{(2)}=6} = 60$$

因为学习率 $\eta = 0.01$ ，所以根据梯度下降法，有

$$w_{11}^{(1)} \leftarrow 10 - 0.01 \times 36 = 9.64, w_{11}^{(2)} \leftarrow 6 - 0.01 \times 60 = 5.4$$

第 2 次迭代：从打印结果可以看出，dropout 层的值等于 0，此时函数 $f = 0$ ，则 f 在 $w_{11}^{(1)} = 9.64$ 和 $w_{11}^{(2)} = 5.4$ 的梯度为

$$\frac{\partial f}{\partial w_{11}^{(1)} | w_{11}^{(1)}=9.64} = 0, \frac{\partial f}{\partial w_{11}^{(2)} | w_{11}^{(2)}=5.4} = 0$$

因为梯度全等于 0，根据梯度下降法， $w_{11}^{(1)}$ 、 $w_{11}^{(2)}$ 的值均不变，即经过第 2 次迭代， $w_{11}^{(1)} = 9.64$ ， $w_{11}^{(2)} = 5.4$ 。

第 3 次迭代：从打印结果可以看出，dropout 层的值不等于 0，此时函数 $f = 6w_{11}^{(1)}w_{11}^{(2)}$ ，在 $w_{11}^{(1)} = 9.64$ ， $w_{11}^{(2)} = 5.4$ 的梯度为

$$\frac{\partial f}{\partial w_{11}^{(1)} | w_{11}^{(1)}=9.64} = 6 \times 5.4 = 32.4, \frac{\partial f}{\partial w_{11}^{(2)} | w_{11}^{(2)}=5.4} = 6 \times 9.24 = 57.84$$

根据梯度下降法，有

$$w_{11}^{(1)} \leftarrow 9.64 - 0.01 \times 32.4 = 9.316, w_{11}^{(2)} \leftarrow 5.4 - 0.01 \times 57.84 = 4.8216$$

第 4 次迭代：从打印结果可以看出，dropout 层的值不等于 0，此时函数 $f = 6w_{11}^{(1)}w_{11}^{(2)}$ ，

$$\frac{\partial f}{\partial w_{11}^{(1)} | w_{11}^{(1)}=9.316} = 6 \times 4.8216 = 28.9296, \frac{\partial f}{\partial w_{11}^{(2)} | w_{11}^{(2)}=4.8216} = 6 \times 9.316 = 55.896,$$

根据梯度下降法，有

$$w_{11}^{(1)} \leftarrow 9.316 - 0.01 \times 28.9296 = 9.026704, w_{11}^{(2)} \leftarrow 4.8216 - 0.01 \times 55.896 = 4.26264$$

依此类推。

AlexNet 取得巨大成功后，网络结构的设计引起了广泛关注，开始出现各种网络结构。接

下来，我们介绍三种经典的网络结构 VGGNet、GoogleNet 和 ResNet。

10.4 VGGNet

VGGNet^[3] 比 AlexNet 的网络层数多，不再使用尺寸较大的卷积核，如 11×11 、 7×7 、 5×5 ，而是只采用尺寸为 3×3 的卷积核。这一点可以从卷积核的分离性理解，如两个 3×3 张量的 full 卷积结果的尺寸为 5×5 ，三个 3×3 张量的 full 卷积结果的尺寸为 7×7 ，而且与两个 3×3 的卷积核依次卷积比直接与一个 5×5 的卷积核卷积的计算量小。图 10-31 所示为 VGG-16 的卷积神经网络结构，其中输入值的尺寸是 224 行 224 列 3 深度。

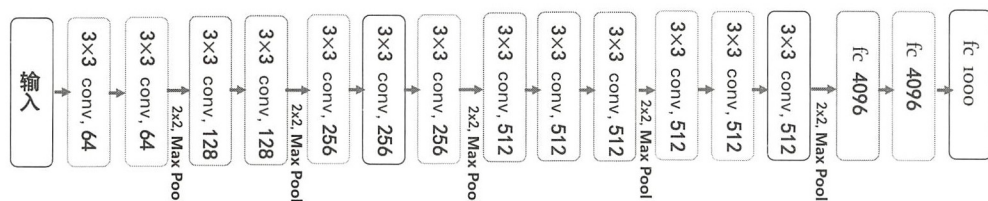


图 10-31 VGG-16 的网络结构

接下来我们利用 TensorFlow 实现该网络结构。

输入层：

```
x=tf.placeholder(tf.float32,[None,224,224,3])
```

第 1 层：输入层与 64 个 3 行 3 列 3 深度的卷积核 same 卷积，将卷积结果加偏置，然后输入 ReLU 激活函数，对应代码如下：

```
with tf.variable_scope('layer1',reuse=tf.AUTO_REUSE):
    # "64个3行3列3深度的卷积核"
    w1 =tf.Variable(tf.random_normal([3,3,3,64]),dtype=tf.float32,name='w')
    # "步长为1的same卷积"
    c1=tf.nn.conv2d(x,w1,[1,1,1,1],'SAME')
    # "因为c1的深度为64，所以偏置的长度为64"
    b1 =tf.Variable(tf.random_normal([64]),dtype=tf.float32,name='b')
    # "卷积结果与偏置相加"
    c1=tf.nn.bias_add(c1,b1)
    # "relu激活函数"
    c1=tf.nn.relu(c1)
```

第2层：将第1层的结果（其尺寸为224行224列64深度）与64个3行3列64深度的卷积核same卷积，将卷积结果加偏置，然后输入ReLU激活函数，对应代码如下：

```
with tf.variable_scope('layer2',reuse=tf.AUTO_REUSE):
    #"64个3行3列64深度的卷积核"
    w2 =tf.Variable(tf.random_normal([3,3,64,64]),dtype=tf.float32,name='w')
    #"步长为1的same卷积"
    c2=tf.nn.conv2d(c1,w2,[1,1,1,1],'SAME')
    #"因为c2的深度为64，所以偏置的长度为64"
    b2 =tf.Variable(tf.random_normal([64]),dtype=tf.float32,name='b')
    #"卷积结果与偏置相加"
    c2=tf.nn.bias_add(c2,b2)
    #"relu激活函数"
    c2=tf.nn.relu(c2)
```

经过第2层后，输出结果的尺寸仍是224行224列64深度，接着对其进行 2×2 的步长为2的最大值池化操作，对应代码如下：

```
#"对c2进行same最大值池化操作，邻域掩码的尺寸为2行2列，步长均为2"
p_c2=tf.nn.max_pool(c2,[1,2,2,1],[1,2,2,1],'SAME')
```

第3层：池化操作后，结果的尺寸为112行112列64深度，也就是将宽和高的尺寸变小了，为了保证每一层的计算量差距不太大，在宽和高方向上虽然尺寸变小了，想办法在深度方向上增加尺寸即可，将池化结果与128个3行3列64深度的卷积核卷积，对应代码如下：

```
with tf.variable_scope('layer3',reuse=tf.AUTO_REUSE):
    #"128个3行3列64深度的卷积核"
    w3 =tf.Variable(tf.random_normal([3,3,64,128]),
                    dtype=tf.float32,name='w')
    #"步长为1的same卷积"
    c3=tf.nn.conv2d(p_c2,w3,[1,1,1,1],'SAME')
    #"因为c3的深度为128，所以偏置的长度为128"
    b3 =tf.Variable(tf.random_normal([128]),dtype=tf.float32,name='b')
    #"卷积结果与偏置相加"
    c3=tf.nn.bias_add(c3,b3)
    #"relu激活函数"
    c3=tf.nn.relu(c3)
```

第4层：经过第3层后，输出结果的尺寸仍为 112 行 112 列 128 深度，接着与 128 个 3 行 3 列 128 深度的卷积核 same 卷积，将卷积结果加偏置，然后输入 ReLU 激活函数，对应代码如下：

```
with tf.variable_scope('layer4',reuse=tf.AUTO_REUSE):
    #"128个3行3列128深度的卷积核"
    w4 =tf.Variable(tf.random_normal([3,3,128,128]),
                    dtype=tf.float32,name='w')
    #"步长为1的same卷积"
    c4=tf.nn.conv2d(c3,w4,[1,1,1,1],'SAME')
    #"因为c4的深度为128，所以偏置的长度为128"
    b4 =tf.Variable(tf.random_normal([128]),dtype=tf.float32,name='b')
    #"卷积结果与偏置相加"
    c4=tf.nn.bias_add(c4,b4)
    #"relu激活函数"
    c4=tf.nn.relu(c4)
```

经过第4层后，输出结果的尺寸为 112 行 112 列 128 深度，接着对其进行 2×2 的步长为 2 的最大值池化操作，对应的代码如下：

```
#"对c4进行same最大值池化操作，邻域掩码的尺寸为2行2列，步长均为2"
p_c4=tf.nn.max_pool(c4,[1,2,2,1],[1,2,2,1],'SAME')
```

第5层：池化操作后的结果的尺寸为 56 行 56 列 128 深度，显然这时宽和高的尺寸减小了，为了在深度方向上增加尺寸，接着与 256 个 3 行 3 列 128 深度卷积核卷积，将卷积结果加偏置，输入 ReLU 激活函数，对应的代码如下：

```
with tf.variable_scope("layer5",reuse=tf.AUTO_REUSE):
    #"256个3行3列128深度的卷积核"
    w5=tf.Variable(tf.random_normal([3,3,128,256]),
                    dtype=tf.float32,name='w')
    #"步长为1的same卷积"
    c5=tf.nn.conv2d(p_c4,w5,[1,1,1,1],'SAME')
    #"因为c5的深度为256，所以偏置的长度为256"
    b5=tf.Variable(tf.random_normal([256]),dtype=tf.float32,name='b')
    #"卷积结果与偏置相加"
    c5=tf.nn.bias_add(c5,b5)
    #"relu激活函数"
    c5=tf.nn.relu(c5)
```

第6层：经过第5层后，输出结果的尺寸为56行56列256深度，接着与256个3行3列256深度的卷积核same卷积，将卷积结果加偏置，然后输入ReLU激活函数，对应的代码如下：

```
with tf.variable_scope("layer6",reuse=tf.AUTO_REUSE):
    #"256个3行3列256深度的卷积核"
    w6 =tf.Variable(tf.random_normal([3,3,256,256]),dtype=tf.float32,name='w')
    #"步长为1的same卷积"
    c6=tf.nn.conv2d(c5,w6,[1,1,1,1],'SAME')
    #"因为c6的深度为256，所以偏置的长度为256"
    b6=tf.Variable(tf.random_normal([256]),dtype=tf.float32,name='b')
    #"卷积结果与偏置相加"
    c6=tf.nn.bias_add(c6,b6)
    #"relu激活函数"
    c6=tf.nn.relu(c6)
```

第7层：经过第6层后，输出结果的尺寸为56行56列256深度，接着与256个3行3列256深度的卷积核same卷积，将卷积结果加偏置，然后输入ReLU激活函数，对应的代码如下：

```
with tf.variable_scope("layer7",reuse=tf.AUTO_REUSE):
    #"256个3行3列256深度的卷积核"
    w7 =tf.Variable(tf.random_normal([3,3,256,256]),
                    dtype=tf.float32,name='w')
    #"步长为1的same卷积"
    c7=tf.nn.conv2d(c6,w7,[1,1,1,1],'SAME')
    #"因为c7的深度为256，所以偏置的长度为256"
    b7=tf.Variable(tf.random_normal([256]),dtype=tf.float32,name='b')
    #"卷积结果与偏置相加"
    c7=tf.nn.bias_add(c7,b7)
    #"relu激活函数"
    c7=tf.nn.relu(c7)
```

经过第7层后，输出结果的尺寸为56行56列256深度，接着对其进行 2×2 的步长为2的最大值池化操作：

```
#"对c7进行same最大值池化操作，邻域掩码的尺寸为2行2列，步长均为2"
p_c7=tf.nn.max_pool(c7,[1,2,2,1],[1,2,2,1],'SAME')
```

第 8 层：经过上述池化操作，输出结果的尺寸为 28 行 28 列 256 深度，显然这时宽和高的尺寸都减小了，为了增加在深度方向上的尺寸，接着与 512 个 3 行 3 列 256 深度的卷积核卷积，将卷积结果加偏置，然后输入 ReLU 激活函数，对应代码如下：

```
with tf.variable_scope("layer8",reuse=tf.AUTO_REUSE):
    # "512个3行3列256深度的卷积核"
    w8 =tf.Variable(tf.random_normal([3,3,256,512]),
                    dtype=tf.float32,name='w')
    # "步长为1的same卷积"
    c8=tf.nn.conv2d(p_c7,w8,[1,1,1,1],'SAME')
    # "因为c8的深度为512，所以偏置的长度为512"
    b8=tf.Variable(tf.random_normal([512]),dtype=tf.float32,name='b')
    # "卷积结果与偏置相加"
    c8=tf.nn.bias_add(c8,b8)
    # "relu激活函数"
    c8=tf.nn.relu(c8)
```

第 9 层：经过第 8 层后，输出结果的尺寸为 28 行 28 列 512 深度，接着与 512 个 3 行 3 列 512 深度的卷积核 same 卷积，将卷积结果加偏置，然后输入 ReLU 激活函数，对应代码如下：

```
with tf.variable_scope("layer9",reuse=tf.AUTO_REUSE):
    # "512个3行3列512深度的卷积核"
    w9=tf.Variable(tf.random_normal([3,3,512,512]),
                    dtype=tf.float32,name='w')
    # "步长为1的same卷积"
    c9=tf.nn.conv2d(c8,w9,[1,1,1,1],'SAME')
    # "因为c9的深度为512，所以偏置的长度为512"
    b9=tf.Variable(tf.random_normal([512]),dtype=tf.float32,name='b')
    # "卷积结果与偏置相加"
    c9=tf.nn.bias_add(c9,b9)
    # "relu激活函数"
    c9=tf.nn.relu(c9)
```

第 10 层：经过第 9 层后，输出结果的尺寸为 28 行 28 列 512 深度，接着与 512 个 3 行 3 列 512 深度的卷积核 same 卷积，将卷积结果加偏置，然后输入 ReLU 激活函数，对应代码如下：


```

with tf.variable_scope("layer10",reuse=tf.AUTO_REUSE):
    #"512个3行3列512深度的卷积核"
    w10=tf.Variable(tf.random_normal([3,3,512,512]),
                    dtype=tf.float32,name='w')
    #"步长为1的same卷积"
    c10=tf.nn.conv2d(c9,w10,[1,1,1,1],'SAME')
    #"因为c10的深度为512，所以偏置的长度为512"
    b10=tf.Variable(tf.random_normal([512]),dtype=tf.float32,name='b')
    #"卷积结果与偏置相加"
    c10=tf.nn.bias_add(c10,b10)
    #"relu激活函数"
    c10=tf.nn.relu(c10)

```

经过第 10 层后，输出结果的尺寸为 28 行 28 列 512 深度，对其进行 2×2 的步长为 2 的最大值池化操作，对应的代码如下：

```
p_c10=tf.nn.max_pool(c10,[1,2,2,1],[1,2,2,1],'SAME')
```

第 11 层：经过上述池化操作后，输出结果的尺寸为 14 行 14 列 512 深度，接着与 512 个 3 行 3 列 512 深度的卷积核卷积，将卷积结果加偏置，然后输入 ReLU 激活函数，对应的代码如下：

```

with tf.variable_scope("layer11",reuse=tf.AUTO_REUSE):
    #"512个3行3列512深度的卷积核"
    w11=tf.Variable(tf.random_normal([3,3,512,512]),
                    dtype=tf.float32,name='w')
    #"步长为1的same卷积"
    c11=tf.nn.conv2d(p_c10,w11,[1,1,1,1],'SAME')
    #"因为c11的深度为512，所以偏置的长度为512"
    b11=tf.Variable(tf.random_normal([512]),dtype=tf.float32,name='b')
    #"卷积结果与偏置相加"
    c11=tf.nn.bias_add(c11,b11)
    #"relu激活函数"
    c11=tf.nn.relu(c11)

```

第 12 层：经过第 11 层后，输出结果的尺寸为 14 行 14 列 512 深度，接着与 512 个 3 行 3 列 512 深度的卷积核 same 卷积，将卷积结果加偏置，然后输入 ReLU 激活函数，对应的代码如下：


```

with tf.variable_scope("layer12",reuse=tf.AUTO_REUSE):
    # "512个3行3列512深度的卷积核"
    w12 =tf.Variable(tf.random_normal([3,3,512,512]),
                     dtype=tf.float32,name='w')
    # "步长为1的same卷积"
    c12=tf.nn.conv2d(c11,w12,[1,1,1,1],'SAME')
    # "因为c12的深度为512，所以偏置的长度为512"
    b12=tf.Variable(tf.random_normal([512]),dtype=tf.float32,name='b')
    # "卷积结果与偏置相加"
    c12=tf.nn.bias_add(c12,b12)
    # "relu激活函数"
    c12=tf.nn.relu(c12)

```

第 13 层：经过第 12 层后，输出结果的尺寸为 14 行 14 列 512 深度，接着与 512 个 3 行 3 列 512 深度的卷积核 same 卷积，将卷积结果加偏置，然后输入 ReLU 激活函数，对应的代码如下：

```

with tf.variable_scope("layer13",reuse=tf.AUTO_REUSE):
    # "512个3行3列512深度的卷积核"
    w13 =tf.Variable(tf.random_normal([3,3,512,512]),
                     dtype=tf.float32,name='w')
    # "步长为1的same卷积"
    c13=tf.nn.conv2d(c12,w13,[1,1,1,1],'SAME')
    # "因为c13的深度为512，所以偏置的长度为512"
    b13=tf.Variable(tf.random_normal([512]),dtype=tf.float32,name='b')
    # "卷积结果与偏置相加"
    c13=tf.nn.bias_add(c13,b13)
    # "relu激活函数"
    c13=tf.nn.relu(c13)

```

经过第 13 层处理后得到的结果尺寸为 14 行 14 列 512 深度，然后对其进行 2×2 的、步长为 2 的最大值池化操作，对应的代码如下：

```

p_c13=tf.nn.max_pool(c13,[1,2,2,1],[1,2,2,1],'SAME')

```

对上述池化操作结果进行拉伸操作，其结果可以看作一个全连接神经网络的输入层，代码如下：

```
shape=p_c13.get_shape()
flatten_p_c13=tf.reshape(p_c13,[-1,shape[1]*shape[2]*shape[3]])
```

第 14 层：作为全连接神经网络的第 1 个隐含层，其神经元的个数为 4096，代码如下：

```
with tf.variable_scope("layer14",reuse=tf.AUTO_REUSE):
    # "权重矩阵和偏置"
    w14 =tf.Variable(
        tf.random_normal([shape[1].value*shape[2].value*shape[3].value,4096]),
        dtype=tf.float32,name='w')
    b14=tf.Variable(tf.random_normal([4096]),dtype=tf.float32,name='b')
    # "线性组合"
    fc14=tf.matmul(flatten_p_c13,w14)
    fc14=tf.nn.bias_add(fc14,b14)
    # "relu线性组合激活"
    fc14=tf.nn.relu(fc14)
```

第 15 层：作为全连接神经网络的第 2 个隐含层，其神经元的个数为 4096，代码如下：

```
with tf.variable_scope("layer15",reuse=tf.AUTO_REUSE):
    # "权重矩阵和偏置"
    w15 =tf.Variable(tf.random_normal([4096,4096]),
        dtype=tf.float32,name='w')
    b15=tf.Variable(tf.random_normal([4096]),dtype=tf.float32,name='b')
    # "线性组合"
    fc15=tf.matmul(fc14,w15)
    fc15=tf.nn.bias_add(fc15,b15)
    # "relu激活函数"
    fc15=tf.nn.relu(fc15)
```

第 16 层：作为全连接神经网络的输出层，其神经元的个数为 4096，代码如下：

```
with tf.variable_scope("layer16",reuse=tf.AUTO_REUSE):
    # "权重矩阵和偏置"
    w16 =tf.Variable(tf.random_normal([4096,1000]),
        dtype=tf.float32,name='w')
    b16=tf.Variable(tf.random_normal([1000]),dtype=tf.float32,name='b')
    # "线性组合"
```

```
fc16=tf.matmul(fc15,w16)
fc16=tf.nn.bias_add(fc16,b16)
```

构建完以上网络结构后，打印 VGGNet 后面的全连接神经网络的输入层的神经元个数，代码如下：

```
print(shape[1].value*shape[2].value*shape[3].value)
```

打印结果为：

```
25088
```

因为第 13 层经过池化操作的尺寸为 7 行 7 列 512 深度，该张量拉伸为一个一维向量的长度为： $7 \times 7 \times 512 = 25088$ 。上述代码加上构造损失函数及其梯度下降的部分，就可以训练一个 VGGNet 了，对 VGG-16 稍加修改就是常提到的 VGG-19 网络，如图 10-32 所示。

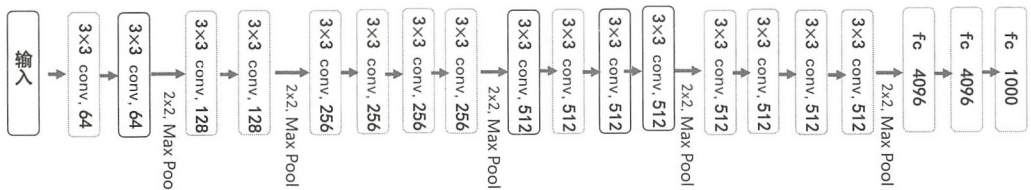


图 10-32 VGG-19 网络结构

10.5 GoogleNet

GoogleNet^[5] 是基于网中网（Network In Network）^[4] 的一种网络结构，以下首先介绍网中网的结构。

10.5.1 网中网结构

我们通过以下示例理解网中网结构。1 个 3 行 3 列 2 深度的张量与 7 个 2 行 2 列 2 深度的卷积核或者 7 个 3 行 3 列 2 深度的卷积核 same 卷积，两者都可以得到的结果的尺寸为 3 行 3 列 7 深度。网中网结构通过多个分支的运算（分支的运算可以为卷积也可以为池化），将分支上的运算结果在深度上连接，得到的结果的尺寸为 3 行 3 列 7 深度，如图 10-33 所示。

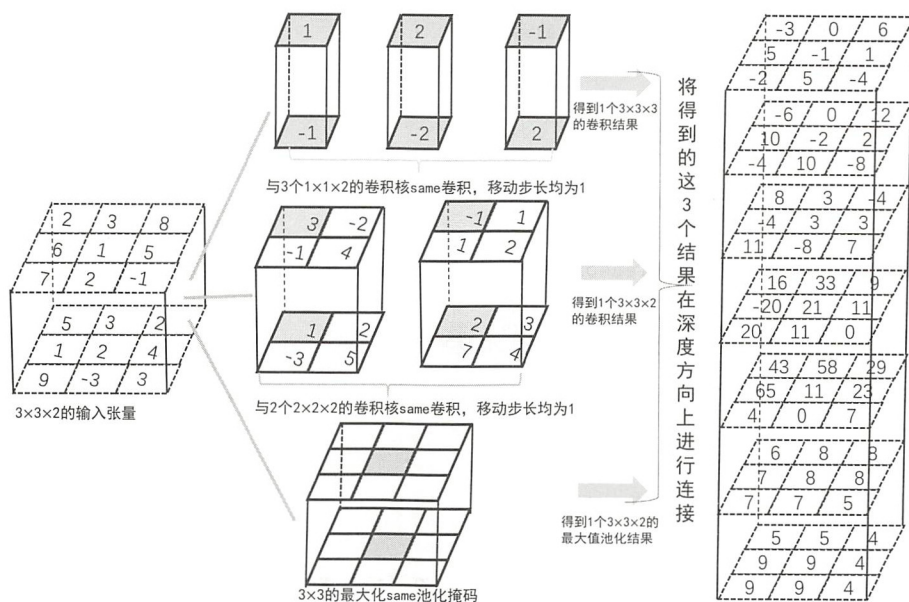


图 10-33 网中网结构

上述示例有3个分支，第1个分支：3行3列2深度的张量先与3个1行1列2深度的卷积核 same 卷积，其结果的尺寸为3行3列3深度。第2个分支：与2个2行2列2深度的卷积核 same 卷积，其结果的尺寸为3行3列2深度。第3个分支：进行3×3的步长为1的 same 最大值池化操作，其结果为3行3列2深度，将这三个分支上的结果在深度的方向上连接，最后的结果为3行3列7深度，上述示例的具体代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
# "1个高为3、宽为3、深度为2(3x3x2)的输入张量"
inputTensor=tf.constant(
    [
        [
            [[2,5],[3,3],[8,2]],
            [[6,1],[1,2],[5,4]],
            [[7,9],[2,-3],[-1,3]]
        ],tf.float32
    )

#
session=tf.Session()
```

```

#"3个高为1、宽为1、深度为2(1×1×2)的卷积核"
filter112_3=tf.constant(
    [
        [[1,2,-1],[-1,-2,2]]
    ],tf.float32
)
result1=tf.nn.conv2d(inputTensor,filter112_3,[1,1,1,1],'SAME')
print(session.run(result1))
#"2个高为2、宽为2、深度为2(2×2×2)的卷积核"
filter222_2=tf.constant(
    [
        [[3,-1],[1,2]], [[-2,1],[2,3]],
        [[-1,1],[-3,7]], [[4,2],[5,4]]
    ],tf.float32)
result2=tf.nn.conv2d(inputTensor,filter222_2,[1,1,1,1],'SAME')
print(session.run(result2))
#"最大值池化"
maxPool_33=tf.nn.max_pool(inputTensor,[1,3,3,1],[1,1,1,1],'SAME')
print(session.run(maxPool_33))
#"深度方向上连接"
result=tf.concat([result1,result2,maxPool_33],3)
print(session.run(result))

```

打印结果如下：

```

[[[[-3.  -6.   8.  16.  43.   6.   5.]
    [ 0.   0.   3.  33.  58.   8.   5.]
    [ 6.  12.  -4.   9.  29.   8.   4.]]

[[ 5.  10.  -4. -20.  65.   7.   9.]
 [-1.  -2.   3.  21.  11.   8.   9.]
 [ 1.   2.   3.  11.  23.   8.   4.]]

[[ -2.  -4.  11.  20.   4.   7.   9.]
 [ 5.  10.  -8.  11.   0.   7.   9.]
 [-4.  -8.   7.   0.   7.   5.   4.]]]

```

GoogleNet 是基于类似网中网模块设计的网络结构，在 GoogleNet 中该模块称为 Inception

Module, 如图 10-34 所示, 多个 Inception Module 模块的组合即为 GoogleNet。

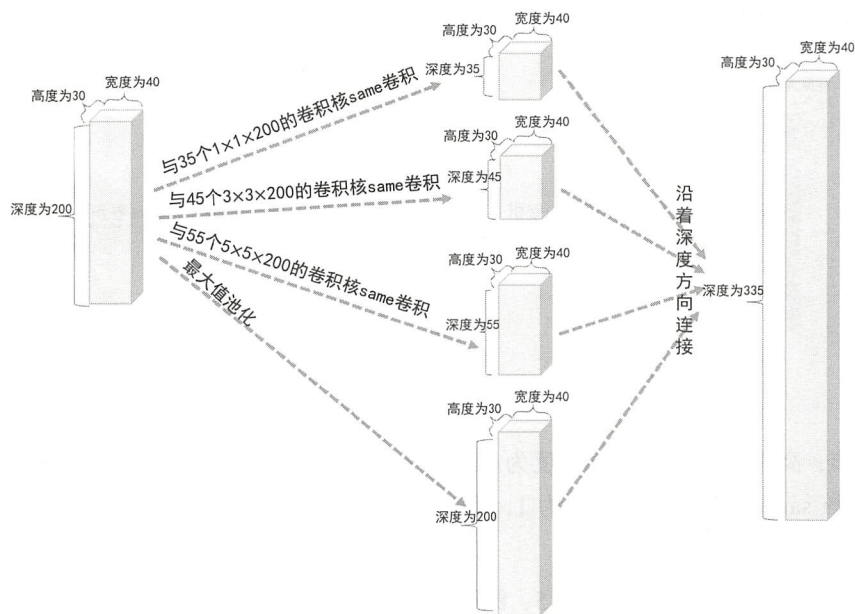


图 10-34 Inception Module

Inception Module 有多种变形, 接下来我们介绍通过 1×1 的卷积核修改的形式。

1. 利用 1×1 的卷积核减少计算成本

假设有 1 个高为 30、宽为 40、深度为 200 的三维张量与 55 个高为 5、宽为 5、深度为 200 的卷积核 same 卷积, 假设移动步长均为 1, 则卷积结果是高为 30、宽为 40、深度为 55 的三维张量, 如图 10-35 所示。

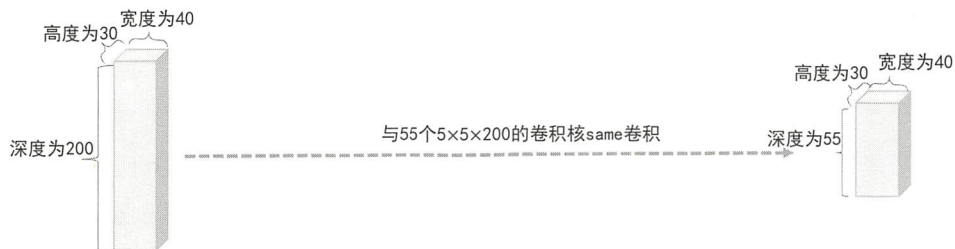


图 10-35 1 个三维张量与 55 个高为 5、宽为 5、深度为 200 的卷积核 same 卷积

该卷积过程的乘法计算量大约为 $5 \times 5 \times 200 \times 30 \times 40 \times 55 = 330000000$ 。

接着，我们考虑以下卷积过程，如图 10-36 所示。

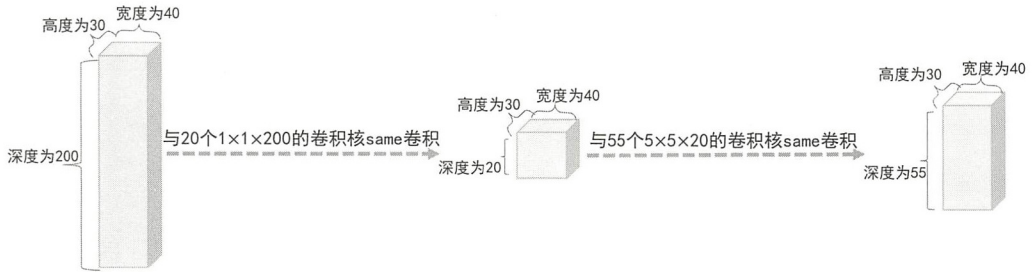


图 10-36 先用 1×1 的卷积核在深度上降维，然后升维

第 1 步：高为 30、宽为 40、深度为 200 的三维张量，先与 20 个高为 1、宽为 1、深度为 20 的卷积核 same 卷积，假设步长为 1，即在深度方向上降维，得到 1 个高为 30、宽为 40、深度为 20 的三维张量。

第 2 步：与 55 个高为 5、宽为 5、深度为 20 的卷积核 same 卷积，移动步长为 1，得到高为 30、宽为 40、深度为 55 的卷积结果。

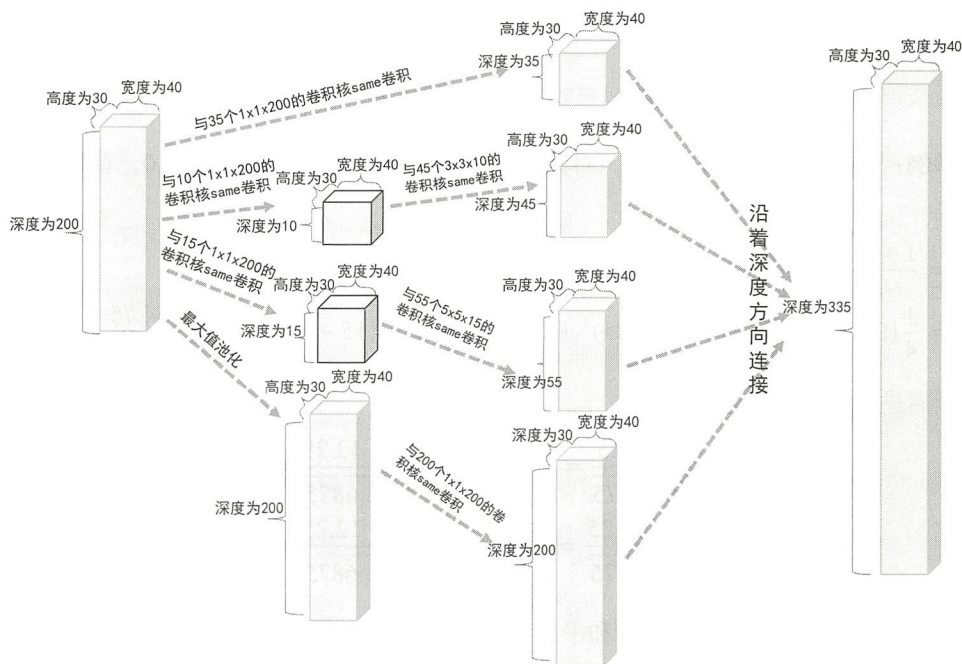
我们来计算上述卷积过程的总乘法计算量。

第 1 步的乘法计算量： $1 \times 1 \times 200 \times 30 \times 40 = 4800000$ 。

第 2 步的乘法计算量： $5 \times 5 \times 20 \times 30 \times 40 \times 55 = 330000000$ 。

总的乘法计算量为 $330000000 + 4800000 = 378000000$ 。

显然，同样是得到高为 30、宽为 40、深度为 55 的卷积结果，采用第 2 种方式，即首先在深度方向上降维，比第 1 种计算方式在计算量上减少了 $\frac{330000000}{378000000} = 8.73015873015873$ 倍。图 10-34 所示的 Inception Module，可以在与 3×3 卷积和 5×5 卷积之前，先利用 1×1 的卷积核降低深度，从而减小计算量，修改后如图 10-37 所示。

图 10-37 利用 1×1 的卷积核的 Inception Module

针对 GoogleNet 的改进, 还需要特别介绍的就是著名的 Batch Normalization^[6] (BN 归一化, 简称 BN), 它有效地解决了在网络层数很深的情况下, 收敛速度很慢的问题, 使用 BN 操作可以加大网络梯度下降的学习率, 加速网络收敛。

10.5.2 Batch Normalization

我们先理解 BN 操作的计算原理, 再介绍其在卷积神经网络中的应用, 假设有数列 $\{x_i\}$, 该数列的 BN 操作过程如下。

先计算数列的均值和方差:

$$\mu = \frac{1}{m} \sum_{i=1}^m x_i, \quad \sigma^2 = \frac{1}{m} \sum_{i=1}^m (x_i - \mu)^2$$

然后, 进行标准化 (normalize) 处理:

$$\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}}, \quad i = 0, 1, 2, \dots, m-1$$

最后，进行放缩和平移，即 BN 操作后的结果：

$$\text{BN}_{\gamma,\beta}(x_i) = \gamma \hat{x}_i + \beta, i = 0, 1, 2, \dots, m-1$$

举例：假设 $\mathbf{x} = (1, 10, 23, 15)$ ，对其进行 BN 操作，假设 $\gamma = 1$ ， $\beta = 0$ ，首先计算均值和方差

$$\mu = \frac{1 + 10 + 23 + 15}{4} = 12.25$$

$$\sigma^2 = \frac{1}{4}\{(1 - 12.25)^2 + (10 - 12.25)^2 + (23 - 12.25)^2 + (15 - 12.25)^2\} = \frac{254.75}{4} = 63.6875$$

然后进行标准化处理，结果为

$$\begin{aligned}\hat{x}_0 &= \frac{1 - 12.25}{\sqrt{63.6875}} = -1.4, & \hat{x}_1 &= \frac{10 - 12.25}{\sqrt{63.6875}} = -0.28 \\ \hat{x}_2 &= \frac{23 - 12.25}{\sqrt{63.6875}} = 1.34, & \hat{x}_3 &= \frac{15 - 12.25}{\sqrt{63.6875}} = 0.34\end{aligned}$$

最后，对归一化结果进行放缩和平移，结果为

$$\text{BN}_{1,0}(\mathbf{x}) = \{-1.4, -0.28, 1.34, 0.34\}$$

TensorFlow 提供函数 `batch_normalization(x, mean, variance, offset, scale, variance_epsilon, name=None)` 实现 BN 操作的功能，其中的参数 `mean` 和参数 `variance` 的值（即均值和方差）是通过利用另一个函数 `moments(x, axes, shift=None)` 进行计算的，利用这两个函数实现以上示例的代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
x=tf.constant([1,10,23,15],tf.float32)
#"计算均值和方差"
mean,variance=tf.nn.moments(x,[0])
#"BatchNormalize"
r=tf.nn.batch_normalization(x,mean,variance,0,1,1e-8)
session=tf.Session()
print(session.run(r))
```

打印结果如下：

```
[-1.4096959 -0.28193915 1.3470427 0.34459233]
```

在利用随机梯度下降训练网络结构时，每次传入网络的是随机抽取的一小批训练数据。我们可以通过以下示例理解卷积神经网络中的 BN 操作，该操作常常是在激活函数之前使用，假设在训练网络时，每次抽取 2 组训练数据，有如图 10-38 所示的 2 个 2 行 2 列 2 深度的三维张量，它是某 2 个样本数据经过某网络的某一卷积层时的结果（注意该结果指的是经过激活函数操作之前的结果），然后分别在每一深度上进行 BN 操作，假设对第 1 深度进行 BN 操作时的 $\gamma = 2$, $\beta = 3$ ；对第 2 深度进行 BN 操作时的 $\gamma = 5$, $\beta = 8$ ，过程如下：

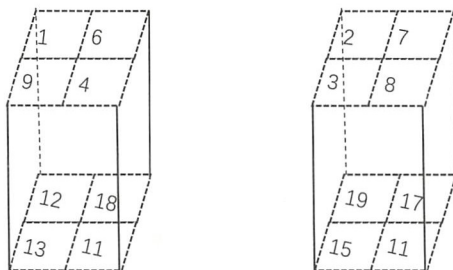


图 10-38 2 个 2 行 2 列 2 深度的三维张量

首先，计算每一深度上的均值和方差，如图 10-39 所示。

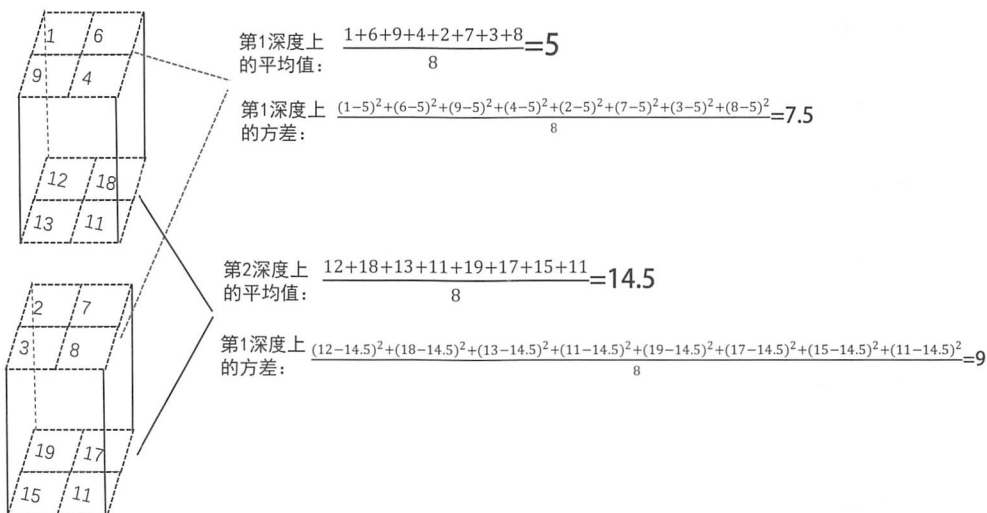


图 10-39 计算每一深度上的均值和方差

然后，对每一深度进行对应的 BN 操作，结果如图 10-40 所示。

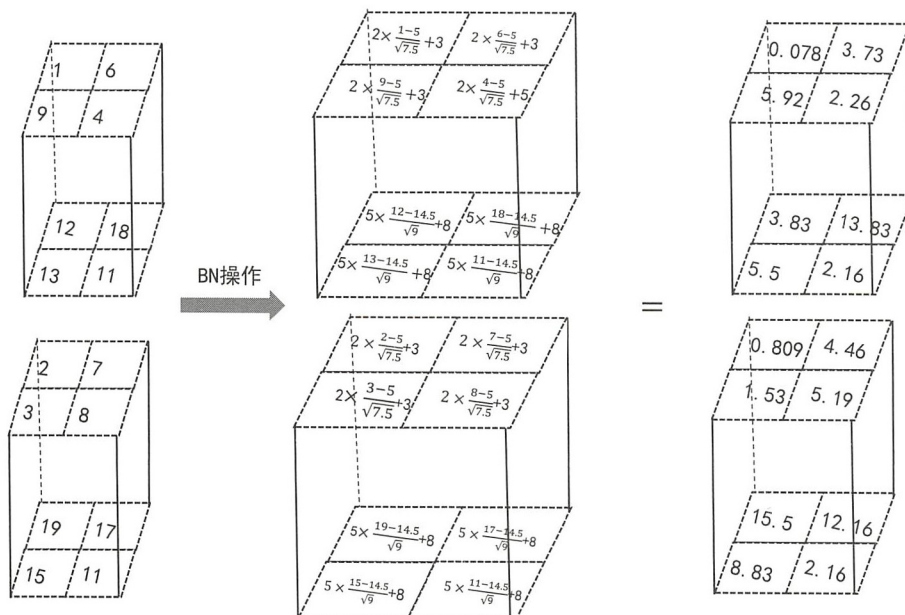


图 10-40 每一个张量在每一深度上根据其均值和方差进行 BN 操作

以上示例对应的代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
# "四维张量"
t=tf.constant([
    # "第1个2行2列2深度的三维张量"
    [
        [[1,12],[6,18]],
        [[9,13],[4,11]],
    ],
    # "第2个2行2列2深度的三维张量"
    [
        [[2,19],[7,17]],
        [[3,15],[8,11]]
    ],tf.float32
)
# "计算均值和方差"moments
```

```

mean,variance=tf.nn.moments(t,[0,1,2])#[0,1,2]
#"BatchNormalize"
gamma=tf.Variable(tf.constant([2,5],tf.float32))
beta=tf.Variable(tf.constant([3,8],tf.float32))
r=tf.nn.batch_normalization(t,mean,variance,beta,gamma,1e-8)
session=tf.Session()
session.run(tf.global_variables_initializer())
#"打印结果"
print('均值和方差:')
print(session.run([mean,variance]))
#"BatchNormalize的结果"
print('BN操作后的结果:')
print(session.run(r))

```

打印结果如下:

```

"均值和方差:"
[array([ 5. , 14.5], dtype=float32), array([7.5, 9. ], dtype=float32)]
"BN操作后的结果:"
[[[ 0.0788132  3.833332 ]
   [ 3.730297 13.833334 ]],
 [[ 5.921187  5.5       ]
   [ 2.2697034 2.166666 ]]],
 [[ 0.8091099 15.5       ]
   [ 4.4605937 12.166666 ]],
 [[ 1.5394068  8.833334 ]
   [ 5.1908903  2.166666 ]]]

```

接下来,我们从另一个角度理解 BN 与卷积运算的关系。

10.5.3 BN 与卷积运算的关系

我们仍以 10.5.2 节中图 10-38 ~ 图 10-40 所示的示例为例,介绍 BN 操作与卷积运算的关系,根据图 10-39 得到两个深度的均值(分别记为 $\mu_1 = 5$, $\mu_2 = 14.5$),方差分别记为 $\sigma_1^2 = 7.5$, $\sigma_2^2 = 9$ 。10.5.2 节中介绍的 BN 操作,标准化步骤为 $\hat{x} = \frac{x-\mu}{\sqrt{\sigma^2}}$,将标准化的步骤换一种写法: $\hat{x} = x \frac{1}{\sqrt{\sigma^2}} + (-\frac{\mu}{\sqrt{\sigma^2}})$,所以图 10-38 所示的两个三维张量的标准化步骤可以用图 10-41 所示的过程表示。

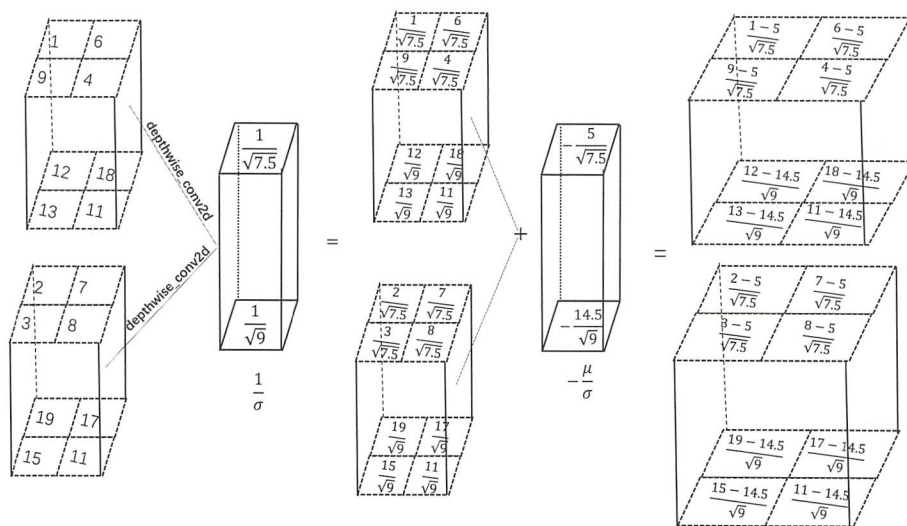


图 10-41 与 $\frac{1}{\sigma}$ 进行 `depthwise_conv2d` 卷积，然后与 $-\frac{\mu}{\sigma}$ 相加

标准化完成之后，再进行平移和缩放，过程如图 10-42 所示。

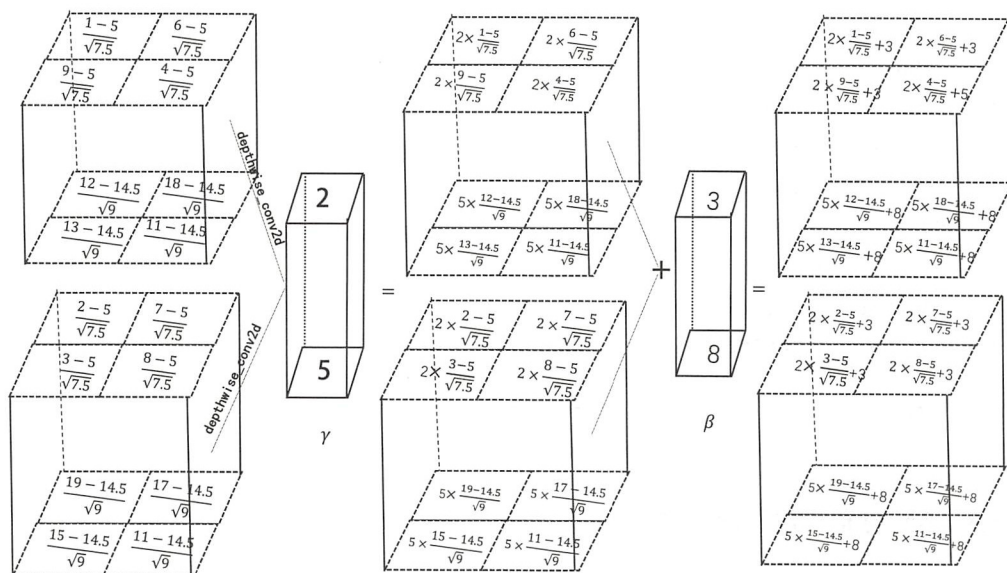


图 10-42 图 10-41 得到的结果与 γ 进行 `depthwise_conv2d` 卷积，然后与 β 相加

所以，对于卷积层中的 BN 操作，可以理解为先进行 `depthwise_conv2d` 卷积，接着与偏置 γ 相加，再进行 `depthwise_conv2d` 卷积，最后再与偏置 β 相加。

训练带有 BN 操作的网络时，对每一深度进行 BN 操作的 γ 和 β 的值和卷积核、偏置类

似，是作为未知数需要进行计算的。假设已经训练好了一个带有 BN 操作的卷积神经网络，但是在使用它进行预测时，往往每次只输入一个样本，那么经过该网络时，计算平均值和方差的意义就不大了，常采用的策略是计算训练阶段的平均值和方差的指数移动平均，然后在预测阶段使用它们作为 BN 操作时的平均值和方差。我们来介绍什么是指数移动平均及对应的 TensorFlow 实现。

10.5.4 指数移动平均

假设变量 x_t 随时间 $t(t \geq 1)$ 变化，按照以下规则定义其指数移动平均值 ($\text{ema}^{(t)}$):

$$\text{ema}^{(t)} = \begin{cases} x_1, & t = 1 \\ \alpha \cdot \text{ema}^{(t-1)} + (1 - \alpha)x_t, & t > 1 \end{cases}$$

示例如下：假设当 $t = 1$ 时， $x_1 = 5$ ，则

$$\text{ema}^{(1)} = x_1 = 5$$

假设当 $t = 2$ 时， $x_2 = 10$ ，则

$$\text{ema}^{(2)} = \alpha \cdot \text{ema}^{(1)} + (1 - \alpha)x_2 = 0.7 \times 5 + (1 - 0.7) \times 10 = 6.5$$

假设当 $t = 3$ 时， $x_3 = 15$ ，则

$$\text{ema}^{(3)} = \alpha \cdot \text{ema}^{(2)} + (1 - \alpha)x_3 = 0.7 \times 6.5 + (1 - 0.7) \times 15 = 9.05$$

假设当 $t = 4$ 时， $x_4 = 20$ ，则

$$\text{ema}^{(4)} = \alpha \cdot \text{ema}^{(3)} + (1 - \alpha)x_4 = 0.7 \times 9.05 + (1 - 0.7) \times 20 = 12.335$$

经过四次运算后，最后的移动平均值为 12.335。

上述示例对应的代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
#"初始化变量，即t=1时的值"
x=tf.Variable(initial_value=5,dtype=tf.float32,trainable=False,name='v')
#"创建计算移动平均的对象"
exp_moving_avg=tf.train.ExponentialMovingAverage(0.7)
update_moving_avg=exp_moving_avg.apply([x])
```

```

#"创建会话"
session=tf.Session()
session.run(tf.global_variables_initializer())
for i in range(4):
    #"打印指数移动平均值"
    session.run(update_moving_avg)
    print('第{}次的移动平均值:'.format(i+1))
    print(session.run(exp_moving_avg.average(x)))
    session.run(x.assign_add(5))

```

打印结果如下：

```

"第1次的移动平均值:"5.0
"第2次的移动平均值:"6.5
"第3次的移动平均值:"9.05
"第4次的移动平均值:"12.335

```

我们已经了解了指数移动平均及 BN 操作，接下来利用 TensorFlow 实现一个简单的带有 BN 操作的卷积神经网络。

10.5.5 带有 BN 操作的卷积神经网络

图 10-43 所示为带有 BN 操作的两层卷积神经网络。

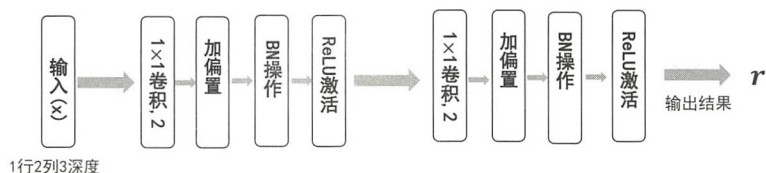


图 10-43 带有 BN 操作的卷积神经网络

其中输入的样本数据的尺寸为 1 行 2 列 3 深度，使用 6.1.2 节介绍的“从 TFRecord 文件中随机解析数据”中的那 3 个三维张量，将这 3 个张量写入文件 dataTest.tfrecord 中，其中第 1 层的卷积核如图 10-44 所示，第 2 层的卷积核如图 10-45 所示。

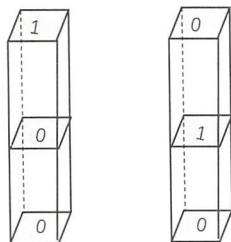


图 10-44 第 1 层的 2 个 1 行 1 列 3 深度的卷积核

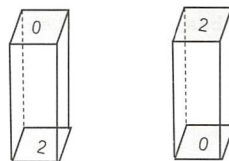


图 10-45 第 2 层的 2 个 1 行 1 列 2 深度的卷积核

所有的偏置均为 0，每次随机从以下代码中解析 2 个张量，作为该卷积神经网络的输入，如何从 TFRecord 文件中随机解析数据已经在 6.1.2 节介绍过，本节不再赘述：

```

#"输入值的占位符"
x=tf.placeholder(tf.float32,[None,1,2,3])
#"设置是否为训练阶段的占位符"
trainable=tf.placeholder(tf.bool)
#"移动平均"
ema=tf.train.ExponentialMovingAverage(0.7)
ema_var_list=[]
#"-----第1层-----"
#"2个1行1列3深度的卷积核"
k1=tf.Variable(tf.constant([
    [[1,0],[0,1],[0,0]],
    ],tf.float32)
)
#"偏置"
b1=tf.Variable(tf.zeros(2))
#"卷积结果加偏置"
c1=tf.nn.conv2d(x,k1,[1,1,1,1],'SAME')+b1
beta1=tf.Variable(tf.zeros(c1.get_shape()[-1].value))
gamma1=tf.Variable(tf.ones(c1.get_shape()[-1].value))
#"计算每一深度上的均值和方差"
m1,v1=tf.nn.moments(c1,[0,1,2])
ema_var_list+= [m1,v1]
#"为了保存均值和方差的指数移动平均"
m1_ema=tf.Variable(tf.zeros(c1.get_shape()[-1]),trainable=False)
v1_ema=tf.Variable(tf.zeros(c1.get_shape()[-1]),trainable=False)

```

```

#"BN操作"
c1_BN=tf.cond(trainable,
               lambda:tf.nn.batch_normalization(c1,m1,v1,beta1,gamma1,1e-8),
               lambda: tf.nn.batch_normalization(c1,m1_ema,
                                                  v1_ema,beta1,gamma1,1e-8)
               )

#"relu激活函数"
r1=tf.nn.relu(c1_BN)
#"-----第2层-----"
#"2个1行1列2深度的卷积核"
k2=tf.Variable(tf.constant(
               [
                 [[2,0],[0,2]]
               ],tf.float32
               ))

#"偏置"
b2=tf.Variable(tf.zeros(2))
#"卷积结果加偏置"
c2=tf.nn.conv2d(r1,k2,[1,1,1,1],'SAME')+b2
beta2=tf.Variable(tf.zeros(c2.get_shape()[-1]))
gamma2=tf.Variable(tf.ones(c2.get_shape()[-1]))
#"计算每一深度上的均值和方差"
m2,v2=tf.nn.moments(c2,[0,1,2])
ema_var_list+=[m2,v2]
#"为了保存均值和方差的指数移动平均"
m2_ema=tf.Variable(tf.zeros(c1.get_shape()[-1]),trainable=False)
v2_ema=tf.Variable(tf.zeros(c1.get_shape()[-1]),trainable=False)
#"BN操作"
c2_BN=tf.cond(trainable,
               lambda:tf.nn.batch_normalization(c2,m2,v2,beta2,gamma2,1e-8),
               lambda:tf.nn.batch_normalization(c2,m2_ema,
                                                  v2_ema,beta2,gamma2,1e-8)
               )

#"relu激活函数"
r2=tf.nn.relu(c2_BN)
update_moving_avg=ema.apply(ema_var_list)

```

```

#"创建会话"
session=tf.Session()
session.run(tf.global_variables_initializer())
session.run(tf.local_variables_initializer())
coord=tf.train.Coordinator()
threads=tf.train.start_queue_runners(sess=session,coord=coord)
num=2
for i in range(num):
    arrs=session.run(arrays)
    print('---第%(num)d批array---'%{'num':i+1})
    print(arrs)
    _,c1_arr=session.run([update_moving_avg,c1],
                          feed_dict={x:arrs,trainable:True})
    print('---第%(num)d次迭代后, 第1个卷积层(卷积结果加偏置)的值---'%{'num':i
+1})
    print(c1_arr)
    #"将计算的指数移动平均的值赋值给Variable对象"
    session.run(m1_ema.assign(ema.average(m1)))
    session.run(v1_ema.assign(ema.average(v1)))
    session.run(m2_ema.assign(ema.average(m2)))
    session.run(v2_ema.assign(ema.average(v2)))
    print('---第%(num)d次迭代后, 第1个卷积层的均值的移动平均值---'%{'num':i+1})
    print(session.run(m1_ema))
coord.request_stop()
coord.join(threads)
session.close()

```

打印结果如下:

```

"---第1批array---"
[[[21. 23. 21.]
  [23. 24. 22.]],
 [[21. 23. 21.]
  [23. 24. 22.]]]
"---第1次迭代后, 第1个卷积层(卷积结果加偏置)的值---"
[[[21. 23.]
  [23. 24.]],

```



```

[[[21. 23.]
  [23. 24.]]]]
"---第1次迭代后，第1个卷积层的均值的移动平均值---"
[6.6000004 7.05      ]
"---第2批array---"
[[[1. 2. 3.]
  [4. 5. 6.]]],
 [[1. 2. 3.]
  [4. 5. 6.]]]
"---第2次迭代后，第1个卷积层(卷积结果加偏置)的值---"
[[[1. 2.]
  [4. 5.]]],
 [[1. 2.]
  [4. 5.]]]
"---第2次迭代后，第1个卷积层的均值的移动平均值---"
[5.3700004 5.985     ]

```

以上代码中打印出了每一次抽取的 2 个三维张量，以及这 2 个张量经过第 1 层卷积的值及其对应的移动平均值。让我们来分析一下打印结果，图 10-46 所示为随机抽取的 2 个三维张量，这 2 个张量与第 1 层的卷积核卷积，加上偏置（即第 1 层激活前的结果），计算其每一深度上的均值。

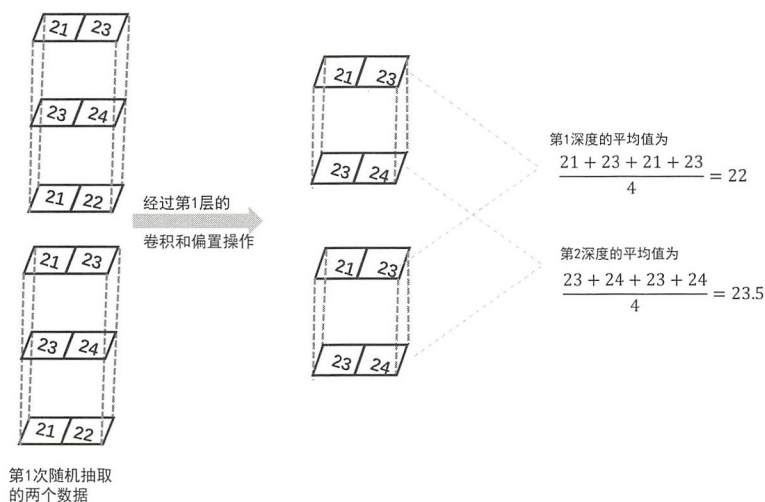


图 10-46 第 1 次迭代

计算第 1 次迭代后，第 1 层均值的移动平均值：

$$0.3 \times \begin{bmatrix} 22 \\ 23.5 \end{bmatrix} = \begin{bmatrix} 6.6 \\ 7.05 \end{bmatrix}$$

图 10-47 所示为第 2 次迭代随机抽取的 2 个三维张量，这 2 个张量与第 1 层的卷积核卷积，加上偏置（即第 1 层激活前的结果），计算其每一深度上的均值。

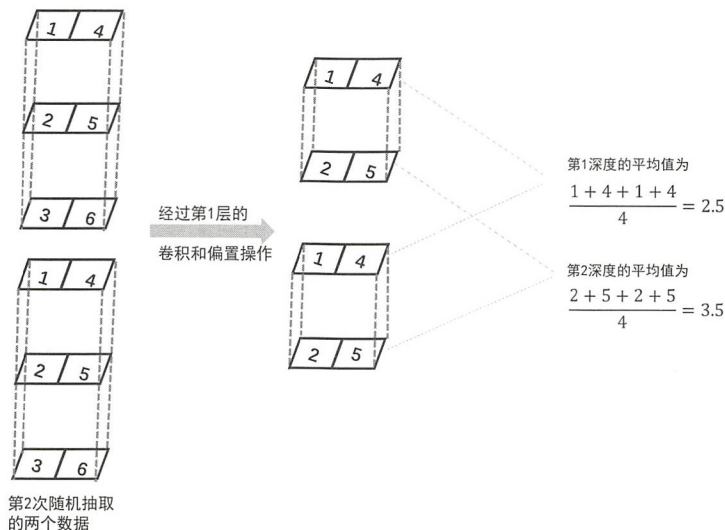


图 10-47 第 2 次迭代

计算第 2 次迭代后，第 1 层均值的移动平均值，依此类推。

$$0.7 \times \begin{bmatrix} 6.6 \\ 7.05 \end{bmatrix} + 0.3 \times \begin{bmatrix} 2.5 \\ 3.5 \end{bmatrix} = \begin{bmatrix} 5.37 \\ 5.985 \end{bmatrix}$$

依此类推。

10.6 ResNet

至此，我们介绍的网络结构本质上都是在加深网络结构。理论上越深的网络在训练集上的准确率越高，但实际上，网络层数增多导致网络退化，即在深层的网络上的准确率不如浅层的，这有些不合理，因为浅层网络的解空间理论上只是深层网络的子集。假设有如图 10-48 所示的网络，该网络在处理某类问题时达到了饱和的准确率。

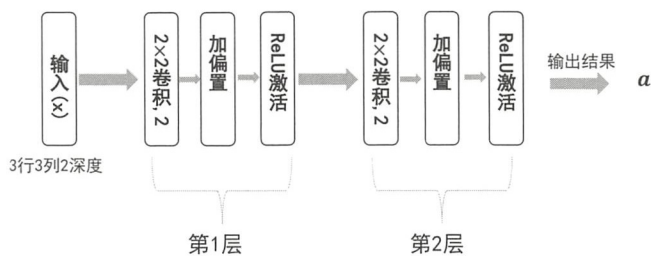


图 10-48 2 层卷积神经网络

修改以上网络结构，在后面追加两层，从 2 层变成 4 层，如图 10-49 所示。

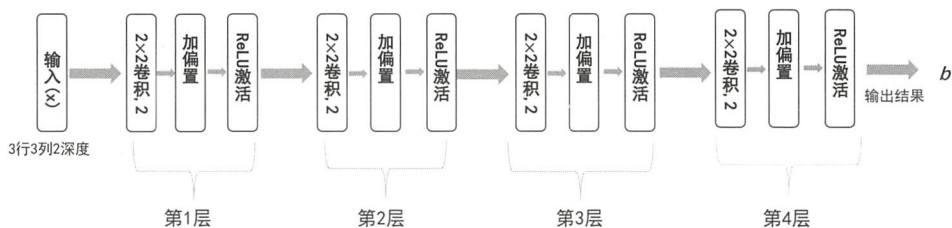


图 10-49 4 层卷积神经网络

前 2 层的卷积核不变，第 3 层和第 4 层的卷积核的尺寸也为 2 行 2 列，我们希望找到第 3 层和第 4 层的卷积核，使得同一个输入 x 经过以上 2 层网络的输出结果 a 和修改后的 4 层网络的输出结果 b 相等，这样起码在深层网络上的表现不会弱于浅层网络，显然这个问题比较难求解。接下来，我们修改图 10-49 所示的 4 层网络，主要是修改第 3 层和第 4 层处的处理方式，如图 10-50 所示。

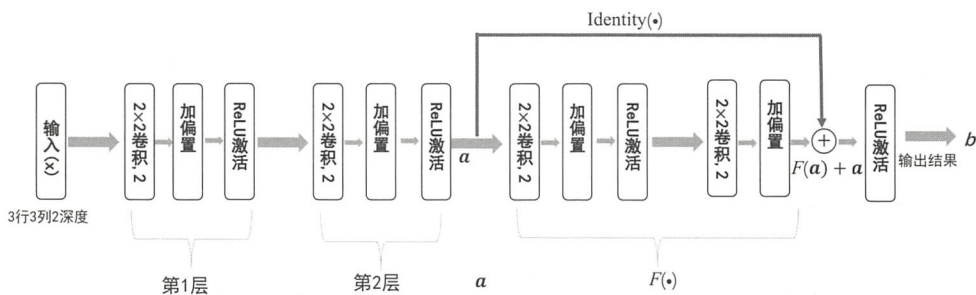


图 10-50 修改图 10-49 所示的卷积神经网络

先单独拿出以上修改的部分，如图 10-51 所示，该模块称为残差模块。

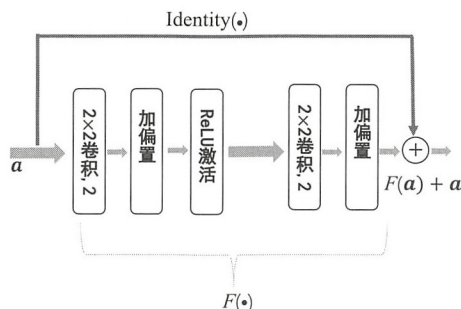


图 10-51 残差模块

我们来介绍该模块是如何计算的。首先，输入 x 经过第 1 层和第 2 层处理后得到 a ， a 经过两个分支路径的处理，一个路径是 Identity 的处理【常称为恒等映射，也称为捷径（shortcut）】，即经过该路径得到的结果还是 a ；另一个路径是 F 的处理，得到的结果记为 $F(a)$ ，然后把两个分支路径的结果相加，即 $a + F(a)$ 。当然， $F(a)$ 和 a 必须尺寸相等才可以相加，这需要通过设计第 3 层和第 4 层的卷积核使得两者的尺寸相等。然后将 $a + F(a)$ 输入 ReLU 激活函数得到最终的输出结果 b ，如何使 b 和 a 相等呢？只要令 $F(a) = 0$ （这里的 0 指的是 3 行 3 列 2 深度的零张量）即可，那么如何令 $F(a) = 0$ 呢？只要让第 3 层和第 4 层的卷积核、偏置均为零张量即可。所以，具备残差模块的网络，使深层网络的表现不弱于浅层网络。

ResNet^[7] 就是具备多个残差模块的网络结构，其中使用的卷积核和 VGGNet 类似，其尺寸均为 3 行 3 列。假设有如图 10-52 所示的残差模块，第 m 层之前的网络的输出值为 a ，其尺寸为 h 行 w 列 64 深度，在经过第 m 层时与 64 个 3 行 3 列的卷积核（当然因为 a 的深度为 64，所以卷积核的深度也为 64）进行步长为 1 的 same 卷积。为了方便讨论，将经过第 m 层的输出结果记为 $a^{(m)}$ ，其尺寸为 h 行 c 列 64 深度； $a^{(m)}$ 经过第 $m+1$ 层的输出结果记为 $a^{(m+1)}$ ，其尺寸为 h 行 c 列 64 深度； $a^{(m+1)}$ 经过第 $m+2$ 层的输出结果记为 $a^{(m+2)}$ ，其尺寸为 h 行 c 列 64 深度，因为 $a^{(m+2)}$ 和 $a^{(m)}$ 的尺寸相等，所以 $a^{(m+2)}$ 经过 shortcut 分支可以直接与 $a^{(m)}$ 相加。

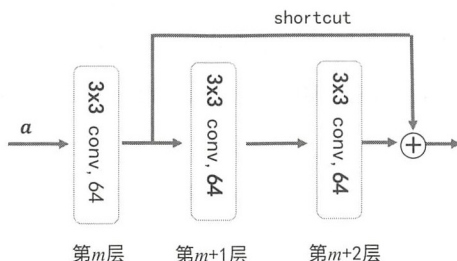


图 10-52 两个分支结果的尺寸相等时的残差模块

图 10-52 所示为两个分支结果的尺寸相等的残差模块。我们来考虑不相等的情况，如图 10-53 所示， $a^{(m)}$ 经过第 $m+1$ 层时与 128 个 3 行 3 列 64 深度的卷积核进行步长为 2 的 same 卷积，经过第 $m+1$ 层时，其输出结果 $a^{(m+1)}$ 的尺寸为 $\text{ceil}(\frac{h}{2})$ 行 $\text{ceil}(\frac{w}{2})$ 列 128 深度。 $a^{(m+1)}$ 经过第 $m+2$ 层时与 128 个 3 行 3 列 128 深度的卷积核进行步长为 1 的 same 卷积，其结果 $a^{(m+2)}$ 的尺寸为 $\text{ceil}(\frac{h}{2})$ 行 $\text{ceil}(\frac{w}{2})$ 列 128 深度。因为 $a^{(m)}$ 与 $a^{(m+2)}$ 的尺寸不相等，所以 $a^{(m)}$ 经过 shortcut 分支是不能直接与 $a^{(m+2)}$ 相加的。常用的方法有两种，一种是对 $a^{(m)}$ 在 shortcut 分支上进行边界补零操作，使它与 $a^{(m+2)}$ 的尺寸相等；另一种是将 $a^{(m)}$ 在 shortcut 分支上与 128 个 1 行 1 列 64 深度的卷积核进行步长为 2 的 same 卷积，这样在 shortcut 分支上的结果的尺寸也为 $\text{ceil}(\frac{h}{2})$ 行 $\text{ceil}(\frac{w}{2})$ 列 128 深度，所以可以与 $a^{(m+2)}$ 相加。

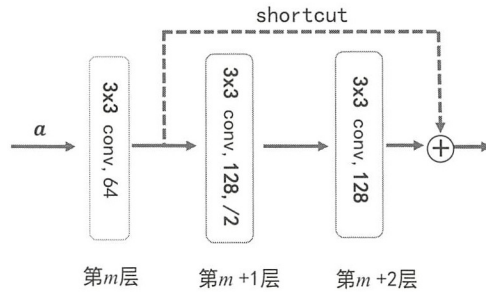


图 10-53 两个分支结果的尺寸不相等时的残差模块

至此，我们已经了解了经典的卷积神经网络结构及其对应的细节。在利用 TensorFlow 训练网络时，我们可以不关注梯度下降实现的具体细节，如全连接神经网络的 BP 算法，直接使用梯度下降函数如 `GradientDescentOptimizer` 或者其他函数即可，但还是有必要了解其具体细节，后续几章将依次介绍训练卷积神经网络时卷积、池化、BN 操作的梯度问题。

10.7 参考文献

- [1] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1989.
- [2] A. Krizhevsky, I. Sutskever, and G. Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*, 2012.
- [3] Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. In: *ICLR*. 2015.
- [4] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. *CoRR*, abs/1312.4400, 2013.

- [5] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pages 1–9, 2015.
- [6] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In Proceedings of The 32nd International Conference on Machine Learning, pages 448–456, 2015.
- [7] He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: CVPR. 2016.

11

卷积的梯度反向传播

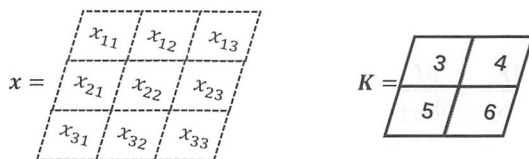
第 10 章介绍的卷积神经网络在处理分类问题时，实际上是利用梯度下降法求解损失函数的最小值问题。既然利用梯度下降法，就需要计算梯度，同 6.4 节介绍的全连接神经网络的梯度反向传播类似，针对神经网络也有对应的梯度反向传播算法，可以有效、快速地计算梯度。下面我们依次介绍关于 valid 卷积和 same 卷积的梯度。

11.1 valid 卷积的梯度

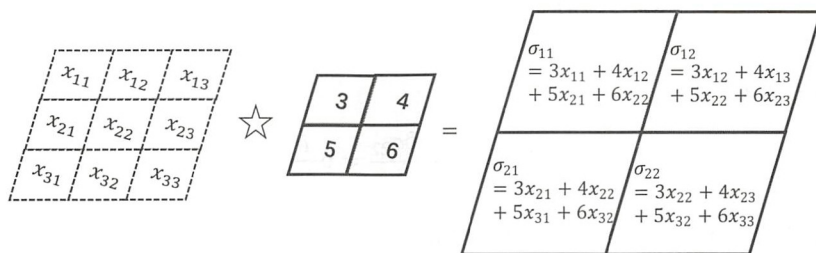
我们分两种不同的情况讨论 valid 卷积的梯度：第一种情况，在已知卷积核的情况下，对未知张量求导（即对张量中每一个变量求导）；第二种情况，在已知张量的情况下，对未知卷积核求导（即对卷积核中每一个变量求导），以下依次讨论这两种情况。

11.1.1 已知卷积核，对未知张量求导

我们用一个简单的例子理解 valid 卷积的梯度反向传播。假设有一个 3×3 的未知张量 \mathbf{x} ，以及已知的 2×2 的卷积核 \mathbf{K} ，如图 11-1 所示。

图 11-1 未知张量 \mathbf{x} 和已知卷积核 \mathbf{K}

假设卷积过程中各方向的移动步长均为 1，则两者 valid 卷积的结果如图 11-2 所示。

图 11-2 \mathbf{x} 和 \mathbf{K} 的 valid 卷积结果

我们考虑以下问题：假设 F 是以上 valid 卷积结果中 σ_{11} 、 σ_{12} 、 σ_{21} 、 σ_{22} 的函数，记为 $F(\sigma_{11}, \sigma_{12}, \sigma_{21}, \sigma_{22})$ ，且 F 是可导的。因为 $\sigma_{ij} (1 \leq i \leq 2, 1 \leq j \leq 2)$ 又是关于 $x_{mn} (1 \leq m \leq 3, 1 \leq n \leq 3)$ 的函数，根据导数的链式法则，计算函数 F 对未知张量中每个变量的导数， F 对 x_{11} 的导数为

$$\frac{\partial F}{\partial x_{11}} = \frac{\partial F}{\partial \sigma_{11}} \frac{\partial \sigma_{11}}{\partial x_{11}} = 3 \frac{\partial F}{\partial \sigma_{11}}$$

F 对 x_{12} 的导数为

$$\frac{\partial F}{\partial x_{12}} = \frac{\partial F}{\partial \sigma_{11}} \frac{\partial \sigma_{11}}{\partial x_{12}} + \frac{\partial F}{\partial \sigma_{12}} \frac{\partial \sigma_{12}}{\partial x_{12}} = 4 \frac{\partial F}{\partial \sigma_{11}} + 3 \frac{\partial F}{\partial \sigma_{12}}$$

F 对 x_{13} 的导数为

$$\frac{\partial F}{\partial x_{13}} = \frac{\partial F}{\partial \sigma_{12}} \frac{\partial \sigma_{12}}{\partial x_{13}} = 4 \frac{\partial F}{\partial \sigma_{12}}$$

F 对 x_{21} 的导数为

$$\frac{\partial F}{\partial x_{21}} = \frac{\partial F}{\partial \sigma_{11}} \frac{\partial \sigma_{11}}{\partial x_{21}} + \frac{\partial F}{\partial \sigma_{21}} \frac{\partial \sigma_{21}}{\partial x_{21}} = 5 \frac{\partial F}{\partial \sigma_{11}} + 3 \frac{\partial F}{\partial \sigma_{21}}$$

F 对 x_{22} 的导数为

$$\frac{\partial F}{\partial x_{22}} = \frac{\partial F}{\partial \sigma_{11}} \frac{\partial \sigma_{11}}{\partial x_{22}} + \frac{\partial F}{\partial \sigma_{12}} \frac{\partial \sigma_{12}}{\partial x_{22}} + \frac{\partial F}{\partial \sigma_{21}} \frac{\partial \sigma_{21}}{\partial x_{22}} + \frac{\partial F}{\partial \sigma_{22}} \frac{\partial \sigma_{22}}{\partial x_{22}} = 6 \frac{\partial F}{\partial \sigma_{11}} + 5 \frac{\partial F}{\partial \sigma_{12}} + 4 \frac{\partial F}{\partial \sigma_{21}} + 3 \frac{\partial F}{\partial \sigma_{22}}$$

F 对 x_{23} 的导数为

$$\frac{\partial F}{\partial x_{23}} = \frac{\partial F}{\partial \sigma_{12}} \frac{\partial \sigma_{12}}{\partial x_{23}} + \frac{\partial F}{\partial \sigma_{22}} \frac{\partial \sigma_{22}}{\partial x_{23}} = 6 \frac{\partial F}{\partial \sigma_{12}} + 4 \frac{\partial F}{\partial \sigma_{22}}$$

F 对 x_{31} 的导数为

$$\frac{\partial F}{\partial x_{31}} = \frac{\partial F}{\partial \sigma_{21}} \frac{\partial \sigma_{21}}{\partial x_{31}} = 5 \frac{\partial F}{\partial \sigma_{21}}$$

F 对 x_{32} 的导数为

$$\frac{\partial F}{\partial x_{32}} = \frac{\partial F}{\partial \sigma_{21}} \frac{\partial \sigma_{21}}{\partial x_{32}} + \frac{\partial F}{\partial \sigma_{22}} \frac{\partial \sigma_{22}}{\partial x_{32}} = 6 \frac{\partial F}{\partial \sigma_{21}} + 5 \frac{\partial F}{\partial \sigma_{22}}$$

F 对 x_{33} 的导数为

$$\frac{\partial F}{\partial x_{33}} = \frac{\partial F}{\partial \sigma_{22}} \frac{\partial \sigma_{22}}{\partial x_{33}} = 6 \frac{\partial F}{\partial \sigma_{22}}$$

仔细观察会发现，以上运算可以利用 full 卷积概括，如图 11-3 所示。

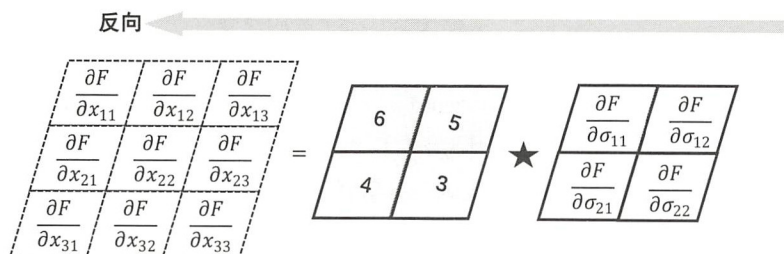


图 11-3 $\frac{\partial F}{\partial \mathbf{x}}$ 和 $\frac{\partial F}{\partial \boldsymbol{\sigma}}$ 的关系

即 $\frac{\partial F}{\partial \mathbf{x}} = \frac{\partial F}{\partial \boldsymbol{\sigma}} \star \text{rotate180}(\mathbf{K})$ ，其中 $\text{rotate180}(\mathbf{K})$ 指 \mathbf{K} 翻转 180°。

理解了导数反向传播，我们通过一个特定的函数 F ，计算 F 在某一点的梯度，假设函数 $F = -\sigma_{11} + \sigma_{12} + 2\sigma_{21} - 2\sigma_{22}$ ，分别计算 F 在 $x_{11} = 1$ ， $x_{12} = 4$ ， $x_{13} = 4$ ， $x_{21} = 2$ ， $x_{22} = 5$ ， $x_{23} = 6$ ， $x_{31} = 9$ ， $x_{32} = 8$ ， $x_{33} = 7$ 处的偏导数，用矩阵表示就是计算 F 在 $\mathbf{x} = \mathbf{x}^{(1)}$ 处的梯度，其中 $\mathbf{x}^{(1)}$ 的值如图 11-4 所示。

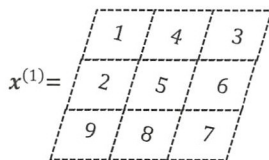


图 11-4 $\mathbf{x}^{(1)}$

首先，计算 F 在 $\boldsymbol{\sigma} = \boldsymbol{\sigma}^{(1)}$ 处的导数，其中 $\boldsymbol{\sigma}^{(1)}$ 的值，如图 11-5 所示。

$$\sigma^{(1)} = \begin{array}{|c|c|c|} \hline 1 & 4 & 3 \\ \hline 2 & 5 & 6 \\ \hline 9 & 8 & 7 \\ \hline \end{array} \star \begin{array}{|c|c|} \hline 3 & 4 \\ \hline 5 & 6 \\ \hline \end{array} = \begin{array}{|c|c|} \hline \sigma_{11}=59 & \sigma_{12}=85 \\ \hline \sigma_{21}=119 & \sigma_{22}=121 \\ \hline \end{array}$$

图 11-5 $\sigma^{(1)}$

F 针对 σ 在 $\sigma = \sigma^{(1)}$ 处的梯度如图 11-6 所示。

$$\frac{\partial F}{\partial \sigma|_{\sigma=\sigma^{(1)}}} = \begin{array}{|c|c|} \hline -1 & 1 \\ \hline 2 & -2 \\ \hline \end{array}$$

图 11-6 F 针对 σ 在 $\sigma = \sigma^{(1)}$ 处的梯度

根据图 11-3 所示的导数关系，我们可以快速地计算 F 在 $\mathbf{x} = \mathbf{x}^{(1)}$ 处的梯度，如图 11-7 所示。

$$\frac{\partial F}{\partial \mathbf{x}|_{\mathbf{x}=\mathbf{x}^{(1)}}} = \frac{\partial F}{\partial \sigma|_{\sigma=\sigma^{(1)}}} \star \text{rotate180}(K) = \begin{array}{|c|c|} \hline -1 & 1 \\ \hline 2 & -2 \\ \hline \end{array} \star \begin{array}{|c|c|} \hline 6 & 5 \\ \hline 4 & 3 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline -3 & -1 & 4 \\ \hline 1 & 1 & -2 \\ \hline 10 & 2 & -12 \\ \hline \end{array}$$

图 11-7 F 在 $\mathbf{x} = \mathbf{x}^{(1)}$ 处梯度

即 F 在 $x_{11} = 1$ 处的偏导数为 -3 ，在 $x_{12} = 4$ 处的偏导数为 -1 ，依此类推。

TensorFlow 提供函数 `tf.nn.conv2d_backprop_input` 实现了 valid 卷积中对未知张量的求导，以上示例对应的实现代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
#"卷积核"
kernel=tf.constant(
    [
        [[3]], [[4]],
        [[5]], [[6]]
    ],tf.float32
)
#"某一函数针对sigma的导数"
out=tf.constant(
    [
        [
```

```

        [[-1],[1]],
        [[2],[-2]]
    ]
    ],tf.float32
)
#"针对未知张量的导数的反向计算"
inputValue=tf.nn.conv2d_backprop_input((1,3,3,1),
                                         kernel,out,[1,1,1,1],'VALID')
#"创建会话"
session=tf.Session()
print(session.run(inputValue))

```

打印结果如下：

```

[[
[[ -3.],[ -1.],[ 4.]],
[[ 1.],[ 1.],[-2.]],
[[ 10.],[ 2.],[-12.]]
]]

```

显然，打印结果与手动计算的结果相同。

推广到一般情况：假设 M 行 N 列的未知张量 \mathbf{x} ， m 行 n 列的卷积核 \mathbf{K} ，两者的 valid 卷积 $\boldsymbol{\sigma} = \mathbf{x} \star \mathbf{K}$ ，如果移动步长为 1，则 $\boldsymbol{\sigma}$ 的尺寸为 $M - m + 1$ 行 $N - n + 1$ 列，函数 F 是关于 σ_{ij} 的函数，且是可导的，其中 $1 \leq i \leq M - m + 1$ ， $1 \leq j \leq N - n + 1$ ，则 $\frac{\partial F}{\partial \boldsymbol{\sigma}}$ 和 $\frac{\partial F}{\partial \mathbf{x}}$ 满足以下 full 卷积关系：

$$\frac{\partial F}{\partial \mathbf{x}} = \frac{\partial F}{\partial \boldsymbol{\sigma}} \star \text{rotate180}(\mathbf{K})$$

其中 $\frac{\partial F}{\partial \mathbf{x}}$ 、 $\frac{\partial F}{\partial \boldsymbol{\sigma}}$ 分别代表 F 对 \mathbf{x} 、 $\boldsymbol{\sigma}$ 的每一个值求导。

具备多深度张量卷积的梯度反向传播也是类似的，因为多深度的卷积实际上是分别在每一个深度上的卷积。

本节讨论的是已知卷积核的情况。11.1.2 节将讨论另一种情况：已知输入张量，卷积核未知的情况。

11.1.2 已知输入张量，对未知卷积核求导

假设已知 3 行 3 列的张量 \mathbf{x} 和未知的 2 行 2 列的卷积核 \mathbf{K} ，如图 11-8 所示。

$$\mathbf{x} = \begin{array}{|c|c|c|} \hline 1 & 2 & 3 \\ \hline 4 & 5 & 6 \\ \hline 7 & 8 & 9 \\ \hline \end{array} \quad \mathbf{K} = \begin{array}{|c|c|} \hline k_{11} & k_{12} \\ \hline k_{21} & k_{22} \\ \hline \end{array}$$

图 11-8 已知张量 \mathbf{x} 和未知卷积核 \mathbf{K}

两者的 valid 卷积结果如图 11-9 所示。

$$\begin{array}{|c|c|c|} \hline 1 & 2 & 3 \\ \hline 4 & 5 & 6 \\ \hline 7 & 8 & 9 \\ \hline \end{array} \star \begin{array}{|c|c|} \hline k_{11} & k_{12} \\ \hline k_{21} & k_{22} \\ \hline \end{array} = \begin{array}{|c|c|} \hline \begin{array}{l} \sigma_{11} \\ = k_{11} + 2k_{12} \\ + 4k_{21} + 5k_{22} \end{array} & \begin{array}{l} \sigma_{12} \\ = 2k_{11} + 3k_{12} \\ + 5k_{21} + 6k_{22} \end{array} \\ \hline \begin{array}{l} \sigma_{21} \\ = 4k_{11} + 5k_{12} \\ + 7k_{21} + 8k_{22} \end{array} & \begin{array}{l} \sigma_{22} \\ = 5k_{11} + 6k_{12} \\ + 8k_{21} + 9k_{22} \end{array} \\ \hline \end{array}$$

图 11-9 \mathbf{x} 和 \mathbf{K} 的 valid 卷积结果

考虑以下问题: 假设 F 是关于 valid 卷积结果中 σ_{11} 、 σ_{12} 、 σ_{21} 、 σ_{22} 的函数, 记为 $F(\sigma_{11}, \sigma_{12}, \sigma_{21}, \sigma_{22})$, 且 F 是可导的。因为 $\sigma_{ij} (1 \leq i \leq 2, 1 \leq j \leq 2)$ 是关于 $k_{mn} (1 \leq m \leq 2, 1 \leq n \leq 2)$ 的函数, 根据导数的链式法则, 计算 F 关于 k_{mn} 的导数, F 关于 k_{11} 的偏导数为

$$\frac{\partial F}{\partial k_{11}} = \frac{\partial F}{\partial \sigma_{11}} \frac{\partial \sigma_{11}}{\partial k_{11}} + \frac{\partial F}{\partial \sigma_{12}} \frac{\partial \sigma_{12}}{\partial k_{11}} + \frac{\partial F}{\partial \sigma_{21}} \frac{\partial \sigma_{21}}{\partial k_{11}} + \frac{\partial F}{\partial \sigma_{22}} \frac{\partial \sigma_{22}}{\partial k_{11}} = \frac{\partial F}{\partial \sigma_{11}} + 2 \frac{\partial F}{\partial \sigma_{21}} + 4 \frac{\partial F}{\partial \sigma_{21}} + 5 \frac{\partial F}{\partial \sigma_{22}}$$

F 关于 k_{12} 的偏导数为

$$\frac{\partial F}{\partial k_{12}} = \frac{\partial F}{\partial \sigma_{11}} \frac{\partial \sigma_{11}}{\partial k_{12}} + \frac{\partial F}{\partial \sigma_{12}} \frac{\partial \sigma_{12}}{\partial k_{12}} + \frac{\partial F}{\partial \sigma_{21}} \frac{\partial \sigma_{21}}{\partial k_{12}} + \frac{\partial F}{\partial \sigma_{22}} \frac{\partial \sigma_{22}}{\partial k_{12}} = 2 \frac{\partial F}{\partial \sigma_{11}} + 3 \frac{\partial F}{\partial \sigma_{21}} + 5 \frac{\partial F}{\partial \sigma_{21}} + 6 \frac{\partial F}{\partial \sigma_{22}}$$

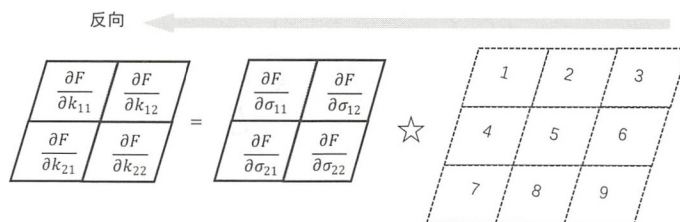
F 关于 k_{21} 的偏导数为

$$\frac{\partial F}{\partial k_{21}} = \frac{\partial F}{\partial \sigma_{11}} \frac{\partial \sigma_{11}}{\partial k_{21}} + \frac{\partial F}{\partial \sigma_{12}} \frac{\partial \sigma_{12}}{\partial k_{21}} + \frac{\partial F}{\partial \sigma_{21}} \frac{\partial \sigma_{21}}{\partial k_{21}} + \frac{\partial F}{\partial \sigma_{22}} \frac{\partial \sigma_{22}}{\partial k_{21}} = 4 \frac{\partial F}{\partial \sigma_{11}} + 5 \frac{\partial F}{\partial \sigma_{21}} + 7 \frac{\partial F}{\partial \sigma_{21}} + 8 \frac{\partial F}{\partial \sigma_{22}}$$

F 关于 k_{22} 的偏导数为

$$\frac{\partial F}{\partial k_{22}} = \frac{\partial F}{\partial \sigma_{11}} \frac{\partial \sigma_{11}}{\partial k_{22}} + \frac{\partial F}{\partial \sigma_{12}} \frac{\partial \sigma_{12}}{\partial k_{22}} + \frac{\partial F}{\partial \sigma_{21}} \frac{\partial \sigma_{21}}{\partial k_{22}} + \frac{\partial F}{\partial \sigma_{22}} \frac{\partial \sigma_{22}}{\partial k_{22}} = 5 \frac{\partial F}{\partial \sigma_{11}} + 6 \frac{\partial F}{\partial \sigma_{21}} + 8 \frac{\partial F}{\partial \sigma_{21}} + 9 \frac{\partial F}{\partial \sigma_{22}}$$

以上导数的运算结果可以用 valid 卷积表示, 如图 11-10 所示。

图 11-10 $\frac{\partial F}{\partial \mathbf{K}}$ 和 $\frac{\partial F}{\partial \boldsymbol{\sigma}}$ 的关系

理解了导数反向传播，我们通过一个特定的函数 F ，计算 F 在某一点的梯度，假设函数 $F = -\sigma_{11} - 2\sigma_{12} - 3\sigma_{21} - 4\sigma_{22}$ ，分别计算 F 在 $k_{11} = 3, k_{12} = 2, k_{21} = 1, k_{22} = 5$ 处的偏导数，用矩阵表示就是 F 在 $\mathbf{K} = \mathbf{K}^{(1)}$ 处的梯度，其中 $\mathbf{K}^{(1)}$ 的值如图 11-11 所示。

$$\mathbf{K}^{(1)} = \begin{bmatrix} 3 & 2 \\ 1 & 5 \end{bmatrix}$$

图 11-11 $\mathbf{K}^{(1)}$

计算 F 针对 $\boldsymbol{\sigma}$ 在 $\boldsymbol{\sigma} = \boldsymbol{\sigma}^{(1)}$ 处的梯度，其中 $\boldsymbol{\sigma}^{(1)}$ 的值如图 11-12 所示。

$$\boldsymbol{\sigma}^{(1)} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \star \begin{bmatrix} 3 & 2 \\ 1 & 5 \end{bmatrix} = \begin{bmatrix} \sigma_{11}=36 & \sigma_{12}=47 \\ \sigma_{21}=69 & \sigma_{22}=80 \end{bmatrix}$$

图 11-12 $\boldsymbol{\sigma}^{(1)}$

显然， F 针对 $\boldsymbol{\sigma}$ 在 $\boldsymbol{\sigma} = \boldsymbol{\sigma}^{(1)}$ 处的梯度如图 11-13 所示。

$$\frac{\partial F}{\partial \boldsymbol{\sigma}}_{|\boldsymbol{\sigma}=\boldsymbol{\sigma}^{(1)}} = \begin{bmatrix} -1 & -2 \\ -3 & -4 \end{bmatrix}$$

图 11-13 F 针对 $\boldsymbol{\sigma}$ 在 $\boldsymbol{\sigma} = \boldsymbol{\sigma}^{(1)}$ 处的梯度

根据图 11-10 所示的导数关系，我们可以快速地计算 F 在 $\mathbf{K}^{(1)}$ 处的梯度（即每一个自变量的偏导数），结果如图 11-14 所示。

$$\frac{\partial F}{\partial \mathbf{K}}_{|\mathbf{K}=\mathbf{K}^{(1)}} = \mathbf{x} \star \frac{\partial F}{\partial \boldsymbol{\sigma}}_{|\boldsymbol{\sigma}=\boldsymbol{\sigma}^{(1)}} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \star \begin{bmatrix} -1 & -2 \\ -3 & -4 \end{bmatrix} = \begin{bmatrix} -37 & -47 \\ -67 & -77 \end{bmatrix}$$

图 11-14 F 在 $\mathbf{K}^{(1)}$ 处的梯度

TensorFlow 提供函数 `tf.nn.conv2d_backprop_filter` 实现 valid 卷积对未知卷积核的求导，以上示例的代码如下：

```

# -*- coding: utf-8 -*-
import tensorflow as tf
#"输入张量"
x=tf.constant(
    [
    [
    [[1],[2],[3]],
    [[4],[5],[6]],
    [[7],[8],[9]]
    ]
    ],
    tf.float32
)
#"某一个函数F对sigma的导数"
partial_sigma=tf.constant(
    [
    [
    [[-1],[-2]],
    [[-3],[-4]]
    ]
    ],tf.float32
)
#"某一个函数F对卷积核k的导数"
partial_sigma_k=tf.nn.conv2d_backprop_filter(x,(2,2,1,1),
        partial_sigma,[1,1,1,1],'VALID')
#"创建会话"
session=tf.Session()
#"打印结果"
print(session.run(partial_sigma_k))

```

打印结果如下：

```

[
[[[-37.]], [[-47.]]],
[[[-67.]], [[-77.]]]
]

```

推广到一般情况，假设有 M 行 N 列的张量 \mathbf{x} ， m 行 n 列的未知卷积核 \mathbf{K} ，假设卷积过程

中各方向的移动步长为 1, 两者的 valid 卷积结果 $\sigma = x \star K$, 显然, σ 的尺寸为 $(M-m+1)$ 行 $(N-n+1)$ 列, 函数 F 是关于 σ_{ij} 的函数, 且是可导的, 其中 $1 \leq i \leq M-m+1, 1 \leq j \leq N-n+1$, 则

$$\frac{\partial F}{\partial K} = x \star \frac{\partial F}{\partial \sigma}$$

其中 $\frac{\partial F}{\partial K}$ 代表 F 分别对 K 中的每一个值求导, $\frac{\partial F}{\partial \sigma}$ 代表 F 分别对 σ 中的每一个值求导。

至此, 我们介绍了关于 valid 卷积的梯度, 接下来介绍 same 卷积的梯度, 仍然讨论两种情况: 对输入张量求导和对卷积核求导。

11.2 same 卷积的梯度

11.2.1 已知卷积核, 对输入张量求导

假设有 3 行 3 列的已知张量 x , 2 行 2 列的未知卷积核 K , 如图 11-15 所示。

$$x = \begin{array}{|c|c|c|} \hline x_{11} & x_{12} & x_{13} \\ \hline x_{21} & x_{22} & x_{23} \\ \hline x_{31} & x_{32} & x_{33} \\ \hline \end{array} \quad K = \begin{array}{|c|c|} \hline 3 & 4 \\ \hline 5 & 6 \\ \hline \end{array}$$

图 11-15 已知张量 x 和未知卷积核 K

两者的 same 卷积结果 σ 如图 11-16 所示。

$$\begin{array}{|c|c|c|} \hline x_{11} & x_{12} & x_{13} \\ \hline x_{21} & x_{22} & x_{23} \\ \hline x_{31} & x_{32} & x_{33} \\ \hline \end{array} \star \begin{array}{|c|c|} \hline 3 & 4 \\ \hline 5 & 6 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline \begin{array}{l} \sigma_{11} \\ = 3x_{11} + 4x_{12} \\ + 5x_{21} + 6x_{22} \end{array} & \begin{array}{l} \sigma_{12} \\ = 3x_{12} + 4x_{13} \\ + 5x_{22} + 6x_{23} \end{array} & \begin{array}{l} \sigma_{13} \\ = 3x_{13} + 5x_{23} \end{array} \\ \hline \begin{array}{l} \sigma_{21} \\ = 3x_{21} + 4x_{22} \\ + 5x_{31} + 6x_{32} \end{array} & \begin{array}{l} \sigma_{22} \\ = 3x_{22} + 4x_{23} \\ + 5x_{32} + 6x_{33} \end{array} & \begin{array}{l} \sigma_{23} \\ = 3x_{23} + 5x_{33} \end{array} \\ \hline \begin{array}{l} \sigma_{31} \\ = 3x_{31} + 4x_{32} \end{array} & \begin{array}{l} \sigma_{32} \\ = 3x_{32} + 4x_{33} \end{array} & \begin{array}{l} \sigma_{33} \\ = 3x_{33} \end{array} \\ \hline \end{array}$$

图 11-16 x 和 K 的 same 卷积

考虑以下问题: 假设 F 是关于 σ_{11} 、 σ_{12} 、 σ_{13} 、 σ_{21} 、 σ_{22} 、 σ_{23} 、 σ_{31} 、 σ_{32} 、 σ_{33} 的函数, 记为 $F(\sigma_{11}, \sigma_{12}, \sigma_{13}, \sigma_{21}, \sigma_{22}, \sigma_{23}, \sigma_{31}, \sigma_{32}, \sigma_{33})$, 且 F 是可导的。因为 σ_{ij} ($1 \leq i \leq 2, 1 \leq j \leq 2$) 又

是关于 x_{mn} ($1 \leq m \leq 3, 1 \leq n \leq 3$) 的函数, 根据导数的链式法则, 计算 F 关于 x_{mn} 的偏导数, F 关于 x_{11} 的偏导数为

$$\frac{\partial F}{\partial x_{11}} = \frac{\partial F}{\partial \sigma_{11}} \frac{\partial \sigma_{11}}{\partial x_{11}} = 3 \frac{\partial F}{\partial \sigma_{11}}$$

F 关于 x_{12} 的偏导数为

$$\frac{\partial F}{\partial x_{12}} = \frac{\partial F}{\partial \sigma_{11}} \frac{\partial \sigma_{11}}{\partial x_{12}} + \frac{\partial F}{\partial \sigma_{12}} \frac{\partial \sigma_{12}}{\partial x_{12}} = 4 \frac{\partial F}{\partial \sigma_{11}} + 3 \frac{\partial F}{\partial \sigma_{12}}$$

F 关于 x_{13} 的偏导数为

$$\frac{\partial F}{\partial x_{13}} = \frac{\partial F}{\partial \sigma_{12}} \frac{\partial \sigma_{12}}{\partial x_{13}} + \frac{\partial F}{\partial \sigma_{13}} \frac{\partial \sigma_{13}}{\partial x_{13}} = 4 \frac{\partial F}{\partial \sigma_{12}} + 3 \frac{\partial F}{\partial \sigma_{13}}$$

F 关于 x_{21} 的偏导数为

$$\frac{\partial F}{\partial x_{21}} = \frac{\partial F}{\partial \sigma_{11}} \frac{\partial \sigma_{11}}{\partial x_{21}} + \frac{\partial F}{\partial \sigma_{21}} \frac{\partial \sigma_{21}}{\partial x_{21}} = 5 \frac{\partial F}{\partial \sigma_{11}} + 3 \frac{\partial F}{\partial \sigma_{21}}$$

F 关于 x_{22} 的偏导数为

$$\frac{\partial F}{\partial x_{22}} = \frac{\partial F}{\partial \sigma_{11}} \frac{\partial \sigma_{11}}{\partial x_{22}} + \frac{\partial F}{\partial \sigma_{12}} \frac{\partial \sigma_{12}}{\partial x_{22}} + \frac{\partial F}{\partial \sigma_{21}} \frac{\partial \sigma_{21}}{\partial x_{22}} + \frac{\partial F}{\partial \sigma_{22}} \frac{\partial \sigma_{22}}{\partial x_{22}} = 6 \frac{\partial F}{\partial \sigma_{11}} + 5 \frac{\partial F}{\partial \sigma_{12}} + 4 \frac{\partial F}{\partial \sigma_{21}} + 3 \frac{\partial F}{\partial \sigma_{22}}$$

F 关于 x_{23} 的偏导数为

$$\frac{\partial F}{\partial x_{23}} = \frac{\partial F}{\partial \sigma_{12}} \frac{\partial \sigma_{12}}{\partial x_{23}} + \frac{\partial F}{\partial \sigma_{13}} \frac{\partial \sigma_{13}}{\partial x_{23}} + \frac{\partial F}{\partial \sigma_{22}} \frac{\partial \sigma_{22}}{\partial x_{23}} + \frac{\partial F}{\partial \sigma_{23}} \frac{\partial \sigma_{23}}{\partial x_{23}} = 6 \frac{\partial F}{\partial \sigma_{12}} + 5 \frac{\partial F}{\partial \sigma_{13}} + 4 \frac{\partial F}{\partial \sigma_{22}} + 3 \frac{\partial F}{\partial \sigma_{23}}$$

F 关于 x_{31} 的偏导数为

$$\frac{\partial F}{\partial x_{31}} = \frac{\partial F}{\partial \sigma_{21}} \frac{\partial \sigma_{21}}{\partial x_{31}} + \frac{\partial F}{\partial \sigma_{31}} \frac{\partial \sigma_{31}}{\partial x_{31}} = 5 \frac{\partial F}{\partial \sigma_{21}} + 3 \frac{\partial F}{\partial \sigma_{31}}$$

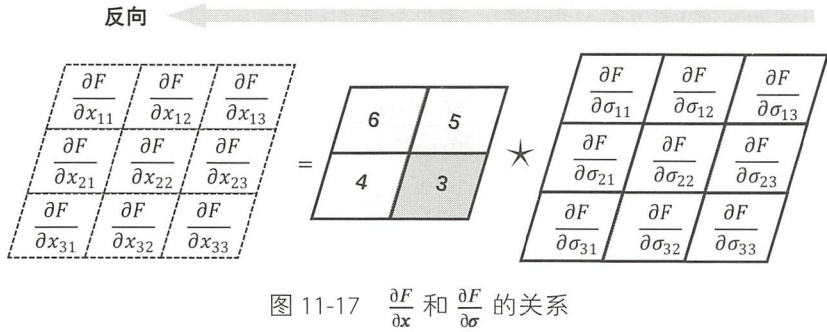
F 关于 x_{32} 的偏导数为

$$\frac{\partial F}{\partial x_{32}} = \frac{\partial F}{\partial \sigma_{21}} \frac{\partial \sigma_{21}}{\partial x_{32}} + \frac{\partial F}{\partial \sigma_{22}} \frac{\partial \sigma_{22}}{\partial x_{32}} + \frac{\partial F}{\partial \sigma_{31}} \frac{\partial \sigma_{31}}{\partial x_{32}} + \frac{\partial F}{\partial \sigma_{32}} \frac{\partial \sigma_{32}}{\partial x_{32}} = 6 \frac{\partial F}{\partial \sigma_{21}} + 5 \frac{\partial F}{\partial \sigma_{22}} + 4 \frac{\partial F}{\partial \sigma_{31}} + 3 \frac{\partial F}{\partial \sigma_{32}}$$

F 关于 x_{33} 的偏导数为

$$\frac{\partial F}{\partial x_{33}} = \frac{\partial F}{\partial \sigma_{22}} \frac{\partial \sigma_{22}}{\partial x_{33}} + \frac{\partial F}{\partial \sigma_{23}} \frac{\partial \sigma_{23}}{\partial x_{33}} + \frac{\partial F}{\partial \sigma_{32}} \frac{\partial \sigma_{32}}{\partial x_{33}} + \frac{\partial F}{\partial \sigma_{33}} \frac{\partial \sigma_{33}}{\partial x_{33}} = 6 \frac{\partial F}{\partial \sigma_{22}} + 5 \frac{\partial F}{\partial \sigma_{23}} + 4 \frac{\partial F}{\partial \sigma_{32}} + 3 \frac{\partial F}{\partial \sigma_{33}}$$

以上运算可以利用以下 same 卷积概括, 如图 11-17 所示。



即 $\frac{\partial F}{\partial \mathbf{x}} = \frac{\partial F}{\partial \boldsymbol{\sigma}} \star \text{rotate180}(\mathbf{K})$ ，其中 $\text{rotate180}(\mathbf{K})$ 指 \mathbf{K} 翻转 180° ，需要注意的是卷积核翻转了 180° ，锚点随着旋转。

假设函数 $F = -\sigma_{11} + \sigma_{12} + 3\sigma_{13} + 2\sigma_{21} - 2\sigma_{22} - 4\sigma_{23} - 3\sigma_{31} + 4\sigma_{32} + \sigma_{33}$ ，分别计算 F 在 $x_{11} = 1, x_{12} = 4, x_{13} = 3, x_{21} = 2, x_{22} = 5, x_{23} = 6, x_{31} = 9, x_{32} = 8, x_{33} = 7$ 处的偏导数，用矩阵表示就是 F 在 $\mathbf{x} = \mathbf{x}^{(1)}$ 处的梯度，其中 $\mathbf{x}^{(1)}$ 的值如图 11-18 所示。

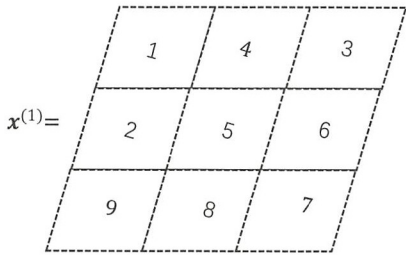


图 11-18 $\mathbf{x}^{(1)}$ 的值

首先，计算 F 针对 $\boldsymbol{\sigma}$ 在 $\boldsymbol{\sigma} = \boldsymbol{\sigma}^{(1)}$ 处的梯度，其中 $\boldsymbol{\sigma}^{(1)}$ 的值如图 11-19 所示。

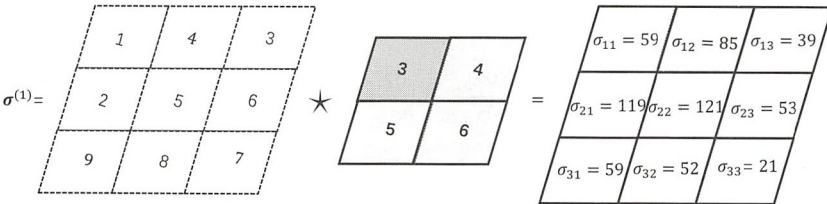


图 11-19 $\boldsymbol{\sigma}^{(1)}$ 的值

显然， F 针对 $\boldsymbol{\sigma}$ 在 $\boldsymbol{\sigma} = \boldsymbol{\sigma}^{(1)}$ 处的梯度如图 11-20 所示。

$$\frac{\partial F}{\partial \sigma|_{\sigma=\sigma^{(1)}}} = \begin{array}{|c|c|c|} \hline -1 & 1 & 3 \\ \hline 2 & -2 & -4 \\ \hline -3 & 4 & 1 \\ \hline \end{array}$$

图 11-20 F 针对 σ 在 $\sigma = \sigma^{(1)}$ 处的梯度

根据图 11-17 所示的导数关系, F 在 $\mathbf{x} = \mathbf{x}^{(1)}$ 处的梯度如图 11-21 所示。

$$\frac{\partial F}{\partial \mathbf{x}|_{\mathbf{x}=\mathbf{x}^{(1)}}} = \frac{\partial F}{\partial \sigma|_{\sigma=\sigma^{(1)}}} \star \text{rotate180}(K) = \begin{array}{|c|c|c|} \hline -1 & 1 & 3 \\ \hline 2 & -2 & -4 \\ \hline -3 & 4 & 1 \\ \hline \end{array} \star \begin{array}{|c|c|} \hline 6 & 5 \\ \hline 4 & 3 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline -3 & -1 & 13 \\ \hline 1 & 1 & 1 \\ \hline 1 & 2 & -13 \\ \hline \end{array}$$

图 11-21 F 在 $\mathbf{x} = \mathbf{x}^{(1)}$ 处的梯度

以上示例的具体代码如下:

```
# -*- coding: utf-8 -*-
import tensorflow as tf
#"卷积核"
kernel=tf.constant(
    [
        [[3]],[[4]]],
        [[5]],[[6]]]
    ],tf.float32
)
#"某一个函数F针对sigma的导数"
partial_sigma=tf.constant(
    [
        [
            [[-1],[1],[3]],
            [[2],[-2],[-4]],
            [[-3],[4],[1]]
        ]
    ],tf.float32
)
#"针对未知张量导数的反向计算"
```



图解深度学习与神经网络：从张量到 TensorFlow 实现

```
partial_x=tf.nn.conv2d_backprop_input((1,3,3,1),kernel,
                                     partial_sigma,[1,1,1,1],'SAME')
#"创建会话"
session=tf.Session()
#"打印结果"
print(session.run(partial_x))
```

打印结果如下：

```
[
[
[[ -3.],[ -1.],[ 13.]],
[[ 1.],[ 1.],[ 1.]],
[[ 1.],[ 2.],[-13.]]
]
```

注意：以上 same 卷积的梯度，卷积核翻转 180° 后锚点的位置规则如下：

- m 为奇数， n 为奇数，锚点的位置是 $\left(\frac{m-1}{2}, \frac{n-1}{2}\right)$
- m 为奇数， n 为偶数，锚点的位置是 $\left(\frac{m-1}{2}, \frac{n}{2}\right)$
- m 为偶数， n 为偶数，锚点的位置是 $\left(\frac{m}{2}, \frac{n}{2}\right)$
- m 为偶数， n 为奇数，锚点的位置是 $\left(\frac{m}{2}, \frac{n-1}{2}\right)$

本节讨论的是已知卷积核的情况，11.2.2 节将反过来讨论另一种情况：已知输入张量，卷积核未知的情况。

11.2.2 已知输入张量，对未知卷积核求导

假设已知 3 行 3 列的张量 \mathbf{x} 和未知的 2 行 2 列的卷积核 \mathbf{K} ，如图 11-22 所示。

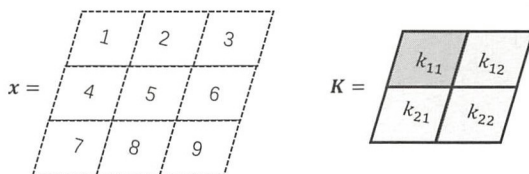


图 11-22 已知张量 \mathbf{x} 和未知卷积核 \mathbf{K}



两者 same 卷积的结果如图 11-23 所示。

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \star \begin{bmatrix} k_{11} & k_{12} \\ k_{21} & k_{22} \end{bmatrix} = \begin{bmatrix} \sigma_{11} = k_{11} + 2k_{12} + 4k_{21} + 5k_{22} & \sigma_{12} = 2k_{11} + 3k_{12} + 5k_{21} + 6k_{22} & \sigma_{13} = 3k_{11} + 6k_{21} \\ \sigma_{21} = 4k_{11} + 5k_{12} + 7k_{21} + 8k_{22} & \sigma_{22} = 5k_{11} + 6k_{12} + 8k_{21} + 9k_{22} & \sigma_{23} = 6k_{11} + 9k_{21} \\ \sigma_{31} = 7k_{11} + 8k_{12} & \sigma_{32} = 8k_{11} + 9k_{12} & \sigma_{33} = 9k_{11} \end{bmatrix}$$

图 11-23 \mathbf{x} 和 \mathbf{K} 的 same 卷积

我们考虑以下问题：假设 F 是关于 σ_{11} 、 σ_{12} 、 σ_{13} 、 σ_{21} 、 σ_{22} 、 σ_{23} 、 σ_{31} 、 σ_{32} 、 σ_{33} 的函数，记为 $F(\sigma_{11}, \sigma_{12}, \sigma_{13}, \sigma_{21}, \sigma_{22}, \sigma_{23}, \sigma_{31}, \sigma_{32}, \sigma_{33})$ ，且 F 是可导的。 $\sigma_{ij} (1 \leq i \leq 3, 1 \leq j \leq 3)$ 又是关于 $k_{mn} (1 \leq m \leq 2, 1 \leq n \leq 2)$ 的函数，那么 $\frac{\partial F}{\partial k}$ 与 $\frac{\partial F}{\partial \sigma}$ 有什么关系呢？如图 11-24 所示。

$$\begin{bmatrix} \frac{\partial F}{\partial k_{11}} & \frac{\partial F}{\partial k_{12}} \\ \frac{\partial F}{\partial k_{21}} & \frac{\partial F}{\partial k_{22}} \end{bmatrix} \longleftrightarrow \begin{bmatrix} \frac{\partial F}{\partial \sigma_{11}} & \frac{\partial F}{\partial \sigma_{12}} & \frac{\partial F}{\partial \sigma_{21}} \\ \frac{\partial F}{\partial \sigma_{21}} & \frac{\partial F}{\partial \sigma_{22}} & \frac{\partial F}{\partial \sigma_{23}} \\ \frac{\partial F}{\partial \sigma_{31}} & \frac{\partial F}{\partial \sigma_{32}} & \frac{\partial F}{\partial \sigma_{33}} \end{bmatrix}$$

图 11-24 $\frac{\partial F}{\partial k}$ 与 $\frac{\partial F}{\partial \sigma}$ 有什么关系呢

根据导数的链式法则，我们可知：

$$\begin{aligned}
 \frac{\partial F}{\partial k_{11}} &= \sum \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} * \begin{bmatrix} \frac{\partial F}{\partial \sigma_{11}} & \frac{\partial F}{\partial \sigma_{12}} & \frac{\partial F}{\partial \sigma_{13}} \\ \frac{\partial F}{\partial \sigma_{21}} & \frac{\partial F}{\partial \sigma_{22}} & \frac{\partial F}{\partial \sigma_{23}} \\ \frac{\partial F}{\partial \sigma_{31}} & \frac{\partial F}{\partial \sigma_{32}} & \frac{\partial F}{\partial \sigma_{33}} \end{bmatrix}, & \frac{\partial F}{\partial k_{12}} &= \sum \begin{bmatrix} 2 & 3 \\ 5 & 6 \\ 8 & 9 \end{bmatrix} * \begin{bmatrix} \frac{\partial F}{\partial \sigma_{11}} & \frac{\partial F}{\partial \sigma_{12}} \\ \frac{\partial F}{\partial \sigma_{21}} & \frac{\partial F}{\partial \sigma_{22}} \\ \frac{\partial F}{\partial \sigma_{31}} & \frac{\partial F}{\partial \sigma_{32}} \end{bmatrix} \\
 \frac{\partial F}{\partial k_{21}} &= \sum \begin{bmatrix} 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} * \begin{bmatrix} \frac{\partial F}{\partial \sigma_{11}} & \frac{\partial F}{\partial \sigma_{12}} & \frac{\partial F}{\partial \sigma_{13}} \\ \frac{\partial F}{\partial \sigma_{21}} & \frac{\partial F}{\partial \sigma_{22}} & \frac{\partial F}{\partial \sigma_{23}} \end{bmatrix}, & \frac{\partial F}{\partial k_{22}} &= \sum \begin{bmatrix} 5 & 6 \\ 8 & 9 \end{bmatrix} * \begin{bmatrix} \frac{\partial F}{\partial \sigma_{11}} & \frac{\partial F}{\partial \sigma_{12}} \\ \frac{\partial F}{\partial \sigma_{21}} & \frac{\partial F}{\partial \sigma_{22}} \end{bmatrix}
 \end{aligned}$$



其中 $*$ 代表张量对应位置相乘， \sum 代表求张量内的值的和，对该问题假设，假设函数

$$F = -\sigma_{11} + \sigma_{12} + 3\sigma_{13} + 2\sigma_{21} - 2\sigma_{22} - 4\sigma_{23} - 3\sigma_{31} + 4\sigma_{32} + \sigma_{33}$$

分别计算 F 在 $k_{11} = 3, k_{12} = 2, k_{21} = 1, k_{22} = 5$ 处的偏导数，用矩阵表示就是 F 在 $\mathbf{K} = \mathbf{K}^{(1)}$ 处的导数，其中 $\mathbf{K}^{(1)}$ 的值如图 11-25 所示。

$$\mathbf{K}^{(1)} =$$

3	2
1	5

图 11-25 $\mathbf{K}^{(1)}$ 的值

首先，计算 F 针对 σ 在 $\sigma = \sigma^{(1)}$ 处的梯度， $\sigma^{(1)}$ 的值如图 11-26 所示。

$$\sigma^{(1)} =$$

1	2	3
4	5	6
7	8	9

$$\star$$

3	2
1	5

$$=$$

$\sigma_{11}=36$	$\sigma_{12}=47$	$\sigma_{13}=15$
$\sigma_{21}=69$	$\sigma_{22}=80$	$\sigma_{23}=27$
$\sigma_{31}=37$	$\sigma_{32}=42$	$\sigma_{33}=27$

图 11-26 $\sigma^{(1)}$ 的值

显然， F 针对 σ 在 $\sigma = \sigma^{(1)}$ 处的梯度，如图 11-27 所示。

$$\frac{\partial F}{\partial \sigma_{|\sigma=\sigma^{(1)}}} =$$

-1	1	3
2	-2	-4
-3	4	1

图 11-27 F 针对 σ 在 $\sigma = \sigma^{(1)}$ 处的梯度

根据导数的链式法则， F 在 $k_{11} = 3, k_{12} = 2, k_{21} = 1, k_{22} = 5$ 处的偏导数如下：

$$\frac{\partial F}{\partial k_{11}} = \sum \left(\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} * \begin{bmatrix} -1 & -2 & 1 \\ -3 & -4 & 2 \\ -2 & 1 & 3 \end{bmatrix} \right) = -1, \quad \frac{\partial F}{\partial k_{12}} = \sum \left(\begin{bmatrix} 2 & 3 \\ 5 & 6 \\ 8 & 9 \end{bmatrix} * \begin{bmatrix} -1 & -2 \\ -3 & -4 \\ -2 & 1 \end{bmatrix} \right) = -54$$



$$\frac{\partial F}{\partial k_{21}} = \sum \left(\begin{bmatrix} 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} * \begin{bmatrix} -1 & -2 & 1 \\ -3 & -4 & 2 \end{bmatrix} \right) = -43$$

$$\frac{\partial F}{\partial k_{22}} = \sum \left(\begin{bmatrix} 5 & 6 \\ 8 & 9 \end{bmatrix} * \begin{bmatrix} -1 & -2 \\ -3 & -4 \end{bmatrix} \right) = -77$$

以上示例的具体代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
#"输入张量"
x=tf.constant(
    [
        [
            [[1],[2],[3]],
            [[4],[5],[6]],
            [[7],[8],[9]]
        ]
    ],
    tf.float32
)
#"某一个函数F对sigma的导数"
partial_sigma=tf.constant(
    [
        [
            [[-1],[-2],[1]],
            [[-3],[-4],[2]],
            [[-2],[1],[3]]
        ]
    ],tf.float32
)
#"某一个函数F对卷积核k的导数"
partial_sigma_k=tf.nn.conv2d_backprop_filter(x,(2,2,1,1),
        partial_sigma,[1,1,1,1],'SAME')
#"创建会话"
session=tf.Session()
#"打印结果"
print(session.run(partial_sigma_k))
```



输出结果如下：

```
[
[[[-1.]], [[-54.]]],
[[[-43.]], [[-77.]]]
]
```

推广到更一般的情况，假设有 M 行 N 列的输入张量 I 及未知的卷积核 k ，函数 F 是关于 I 和 k same 卷积 σ 的函数，那么 F 对未知的卷积核 k 中每一个变量的导数的规则如下。

在卷积核中距离锚点右侧第 R 个、下侧第 B 个位置的未知数的导数等于：

$$I[B+1:M][R+1:N] * \frac{\partial F}{\partial \sigma}[1:M-B][1:N-R]$$

即 I 的第 $B+1$ 至 M 行，第 $R+1$ 至 N 列， $\frac{\partial F}{\partial \sigma}$ 的第 1 至 $M-B$ 行，第 1 至 $N-R$ 列，对应位置相乘的和。

在卷积核中距离锚点右侧第 R 个、上侧第 T 个位置的未知数的导数等于：

$$I[1:M-T][R+1:N] * \frac{\partial F}{\partial \sigma}[T+1:M][1:N-R]$$

即 I 的第 1 至 $M-T$ 行，第 $R+1$ 至 N 列， $\frac{\partial F}{\partial \sigma}$ 的第 $T+1$ 至 M 行，第 1 至 $N-R$ 列，对应位置相乘的和。

在卷积核中距离锚点左侧第 L 个、上侧第 T 个位置的未知数的导数等于：

$$I[1:M-T][1:N-L] * \frac{\partial F}{\partial \sigma}[T+1:M][L+1:N]$$

即 I 的第 1 至 $M-T$ 行，第 1 至 $N-L$ 列， $\frac{\partial F}{\partial \sigma}$ 的第 $T+1$ 至 M 行，第 $L+1$ 至 N 列，对应位置相乘的和。

在卷积核中距离锚点左侧第 L 个、下侧第 B 个位置的未知数的导数等于：

$$I[B+1:M][1:N-L] * \frac{\partial F}{\partial \sigma}[1:M-B][1:N-L]$$

即 I 的第 $B+1$ 至 M 行，第 1 至 $N-L$ 列， $\frac{\partial F}{\partial \sigma}$ 的第 1 至 $M-B$ 行，第 1 至 $N-L$ 列，对应位置相乘的和。

在第 12 章，我们将介绍关于池化操作的梯度计算。



12

池化操作的梯度

池化操作的梯度分两部分介绍，第一部分介绍平均值池化的梯度计算，第二部分介绍最大值池化的梯度计算。

12.1 平均值池化的梯度

我们通过以下示例理解平均值池化的梯度计算，假设有 9 个自变量 x_{mn} ，其中 $1 \leq m \leq 3$ ， $1 \leq n \leq 3$ ， σ_{ij} 与 x_{mn} 有如图 12-1 所示的关系，其中 $1 \leq i \leq 2$ ， $1 \leq j \leq 2$ 。

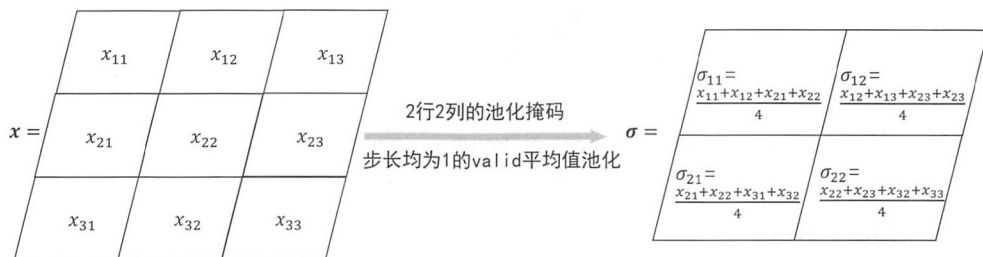


图 12-1 x_{mn} 与 σ_{ij} 的关系

σ 是 x 经过 valid 平均值池化的结果，其中邻域掩码为 2 行 2 列，行和列方向的移动步长均为 1。



考虑以下问题：假设 F 是关于 $\sigma_{ij}, 1 \leq i \leq 2, 1 \leq j \leq 2$ 的函数，又因为 σ_{ij} 是关于 $x_{mn}, 1 \leq m \leq 3, 1 \leq n \leq 3$ 的函数，所以 F 是关于 x_{mn} 的复合函数，考虑一个比较简单的函数：

$$F = \sigma_{11} + \sigma_{12} + \sigma_{21} + \sigma_{22}$$

根据导数的链式法则，计算 F 关于 x_{mn} 的导数， F 关于 x_{11} 的偏导数为

$$\frac{\partial F}{\partial x_{11}} = \frac{\partial F}{\partial \sigma_{11}} \frac{\partial \sigma_{11}}{\partial x_{11}} = \frac{1}{4} \frac{\partial F}{\partial \sigma_{11}} = \frac{1}{4}$$

F 关于 x_{12} 的偏导数为

$$\frac{\partial F}{\partial x_{12}} = \frac{\partial F}{\partial \sigma_{11}} \frac{\partial \sigma_{11}}{\partial x_{12}} + \frac{\partial F}{\partial \sigma_{12}} \frac{\partial \sigma_{12}}{\partial x_{12}} = \frac{1}{4} \frac{\partial F}{\partial \sigma_{11}} + \frac{1}{4} \frac{\partial F}{\partial \sigma_{12}} = \frac{1}{2}$$

F 关于 x_{13} 的偏导数为

$$\frac{\partial F}{\partial x_{13}} = \frac{\partial F}{\partial \sigma_{12}} \frac{\partial \sigma_{12}}{\partial x_{13}} = \frac{1}{4} \frac{\partial F}{\partial \sigma_{12}} = \frac{1}{4}$$

F 关于 x_{21} 的偏导数为

$$\frac{\partial F}{\partial x_{21}} = \frac{\partial F}{\partial \sigma_{11}} \frac{\partial \sigma_{11}}{\partial x_{21}} + \frac{\partial F}{\partial \sigma_{21}} \frac{\partial \sigma_{21}}{\partial x_{21}} = \frac{1}{4} \frac{\partial F}{\partial \sigma_{11}} + \frac{1}{4} \frac{\partial F}{\partial \sigma_{21}} = \frac{1}{2}$$

F 关于 x_{22} 的偏导数为

$$\frac{\partial F}{\partial x_{22}} = \frac{\partial F}{\partial \sigma_{11}} \frac{\partial \sigma_{11}}{\partial x_{22}} + \frac{\partial F}{\partial \sigma_{12}} \frac{\partial \sigma_{12}}{\partial x_{22}} + \frac{\partial F}{\partial \sigma_{21}} \frac{\partial \sigma_{21}}{\partial x_{22}} + \frac{\partial F}{\partial \sigma_{22}} \frac{\partial \sigma_{22}}{\partial x_{22}} = 1$$

F 关于 x_{23} 的偏导数为

$$\frac{\partial F}{\partial x_{23}} = \frac{\partial F}{\partial \sigma_{12}} \frac{\partial \sigma_{12}}{\partial x_{23}} + \frac{\partial F}{\partial \sigma_{22}} \frac{\partial \sigma_{22}}{\partial x_{23}} = \frac{1}{4} \frac{\partial F}{\partial \sigma_{12}} + \frac{1}{4} \frac{\partial F}{\partial \sigma_{22}} = \frac{1}{2}$$

F 关于 x_{31} 的偏导数为

$$\frac{\partial F}{\partial x_{31}} = \frac{\partial F}{\partial \sigma_{21}} \frac{\partial \sigma_{21}}{\partial x_{31}} = \frac{1}{4} \frac{\partial F}{\partial \sigma_{21}} = \frac{1}{4}$$

F 关于 x_{32} 的偏导数为

$$\frac{\partial F}{\partial x_{32}} = \frac{\partial F}{\partial \sigma_{21}} \frac{\partial \sigma_{21}}{\partial x_{32}} + \frac{\partial F}{\partial \sigma_{22}} \frac{\partial \sigma_{22}}{\partial x_{32}} = \frac{1}{4} \frac{\partial F}{\partial \sigma_{21}} + \frac{1}{4} \frac{\partial F}{\partial \sigma_{22}} = \frac{1}{2}$$

F 关于 x_{33} 的偏导数为

$$\frac{\partial F}{\partial x_{33}} = \frac{\partial F}{\partial \sigma_{22}} \frac{\partial \sigma_{22}}{\partial x_{33}} = \frac{1}{4} \frac{\partial F}{\partial \sigma_{22}} = \frac{1}{4}$$

我们利用计算梯度的函数 `gradients` 实现上述示例，具体代码如下：

```
# -*- coding: utf-8 -*-
```



```

import tensorflow as tf
import numpy as np
#"x是1个3行3列1深度的张量"
x=tf.placeholder(tf.float32,(1,3,3,1))
#"2x2的掩码，步长是(1,1,1,1)的valid平均值池化操作"
sigma=tf.nn.avg_pool(x,(1,2,2,1),(1,1,1,1),'VALID')
#"利用上述池化操作的结果，构造一个函数F:池化结果的和"
F=tf.reduce_sum(sigma)
#"创建会话"
session=tf.Session()
#"分别计算F在某一点xvalue处针对sigma和x的梯度"
xvalue=np.random.randn(1,3,3,1)
grad=tf.gradients(F,[sigma,x])
results=session.run(grad,{x:xvalue})
#"打印结果"
print("----针对sigma的梯度----:")
print(results[0])
print("----针对x的梯度----:")
print(results[1])

```

打印结果如下：

```
"----针对sigma的梯度----:"
```

```

[[[ [ 1.]
      [ 1.]]

  [[ 1.]
   [ 1.]]]]

```

```
"----针对x的梯度----:"
```

```

[[[ [ 0.25]
      [ 0.5 ]
      [ 0.25]]

  [[ 0.5 ]
   [ 1.  ]
   [ 0.5 ]]]

```

图解深度学习与神经网络：从张量到 TensorFlow 实现

```
[[ 0.25]
 [ 0.5 ]
 [ 0.25]]]]
```

打印结果与手动计算的结果相等。12.2 节将介绍关于最大值池化的梯度。

12.2 最大值池化的梯度

我们通过以下示例理解最大值池化的梯度，假设有 16 个自变量 x_{mn} ，其中 $1 \leq m \leq 4$ ， $1 \leq n \leq 4$ ， σ_{ij} 与 x_{mn} 有如图 12-2 所示的关系，其中 $1 \leq i \leq 2$ ， $1 \leq j \leq 2$ 。

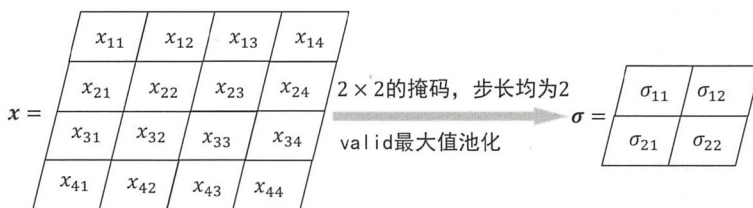


图 12-2 σ_{ij} 与 x_{mn} 的关系

根据 valid 最大值池化的定义得到

$$\sigma_{11} = \max(x_{11}, x_{12}, x_{21}, x_{22}), \sigma_{12} = \max(x_{13}, x_{14}, x_{23}, x_{24})$$

$$\sigma_{21} = \max(x_{31}, x_{32}, x_{41}, x_{42}), \sigma_{22} = \max(x_{33}, x_{34}, x_{43}, x_{44})$$

假设函数 F 是关于 σ_{11} 、 σ_{12} 、 σ_{21} 、 σ_{22} 的函数，且 $F = \sigma_{11}^2 + \sigma_{12}^2 + \sigma_{21}^2 + \sigma_{22}^2$ ，我们利用标准的梯度下降法求解最小值问题 $\min F$ 。

首先，初始化 \mathbf{x} 的值为 $\mathbf{x}^{(1)}$ ，如图 12-3 所示。

$$\mathbf{x}^{(1)} =$$

8	2	9	3
4	6	7	10
20	13	1	5
12	18	19	14

图 12-3 $\mathbf{x}^{(1)}$ 的值

学习率 $\eta = 0.5$ 。第 1 次迭代：针对 $\mathbf{x}^{(1)}$ 的最大值池化的结果为

$$\sigma_{11} = x_{11} = 8, \sigma_{12} = x_{24} = 10, \sigma_{21} = x_{31} = 20, \sigma_{22} = x_{43} = 19$$

这时函数 F 只与 x_{11} 、 x_{24} 、 x_{31} 、 x_{43} 有关, 即 $F = x_{11}^2 + x_{24}^2 + x_{31}^2 + x_{43}^2$, 计算 F 在 $\mathbf{x}^{(1)}$ 处的梯度, 以下依次计算 F 在 $\mathbf{x}^{(1)}$ 处每一个自变量的偏导数:

$$\begin{aligned} \frac{\partial F}{\partial x_{11} | x_{11}=8} &= 2 \times x_{11} = 2 \times 8 = 16, & \frac{\partial F}{\partial x_{12} | x_{12}=2} &= \frac{\partial F}{\partial x_{21} | x_{21}=4} = \frac{\partial F}{\partial x_{22} | x_{22}=6} = 0 \\ \frac{\partial F}{\partial x_{24} | x_{24}=10} &= 2 \times x_{24} = 2 \times 10 = 20, & \frac{\partial F}{\partial x_{13} | x_{13}=9} &= \frac{\partial F}{\partial x_{14} | x_{14}=3} = \frac{\partial F}{\partial x_{23} | x_{23}=7} = 0 \\ \frac{\partial F}{\partial x_{31} | x_{31}=20} &= 2 \times x_{31} = 2 \times 20 = 40, & \frac{\partial F}{\partial x_{32} | x_{32}=13} &= \frac{\partial F}{\partial x_{41} | x_{41}=12} = \frac{\partial F}{\partial x_{42} | x_{42}=18} = 0 \\ \frac{\partial F}{\partial x_{43} | x_{43}=19} &= 2 \times x_{43} = 2 \times 19 = 38, & \frac{\partial F}{\partial x_{33} | x_{33}=1} &= \frac{\partial F}{\partial x_{34} | x_{34}=5} = \frac{\partial F}{\partial x_{44} | x_{44}=14} = 0 \end{aligned}$$

即 F 在 $\mathbf{x} = \mathbf{x}^{(1)}$ 处的梯度, 如图 12-4 所示。

$$\frac{\partial F}{\partial \mathbf{x} | \mathbf{x}=\mathbf{x}^{(1)}} = \begin{array}{|c|c|c|c|} \hline 16 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 20 \\ \hline 40 & 0 & 0 & 0 \\ \hline 0 & 0 & 38 & 0 \\ \hline \end{array}$$

图 12-4 F 在 $\mathbf{x} = \mathbf{x}^{(1)}$ 处的梯度

根据标准的梯度下降法, 结果如下:

$$\mathbf{x}^{(2)} = \mathbf{x}^{(1)} - \eta \frac{\partial F}{\partial \mathbf{x} | \mathbf{x}=\mathbf{x}^{(1)}}$$

展开后如图 12-5 所示。

$$\mathbf{x}^{(2)} = \begin{array}{|c|c|c|c|} \hline 8 & 2 & 9 & 3 \\ \hline 4 & 6 & 7 & 10 \\ \hline 20 & 13 & 1 & 5 \\ \hline 12 & 18 & 19 & 14 \\ \hline \end{array} - 0.5 \times \begin{array}{|c|c|c|c|} \hline 16 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 20 \\ \hline 40 & 0 & 0 & 0 \\ \hline 0 & 0 & 38 & 0 \\ \hline \end{array} = \begin{array}{|c|c|c|c|} \hline 0 & 2 & 9 & 3 \\ \hline 4 & 6 & 7 & 0 \\ \hline 0 & 13 & 1 & 5 \\ \hline 12 & 18 & 0 & 14 \\ \hline \end{array}$$

图 12-5 利用梯度下降法计算的第 1 次迭代值

第 2 次迭代: 对 $\mathbf{x}^{(2)}$ 进行最大值池化操作, 结果如下:

$$\sigma_{11} = x_{22} = 6, \sigma_{12} = x_{13} = 9, \sigma_{21} = x_{42} = 18, \sigma_{22} = x_{44} = 14$$

函数 F 只与 x_{22} 、 x_{13} 、 x_{42} 、 x_{44} 有关, 即 $F = x_{22}^2 + x_{13}^2 + x_{42}^2 + x_{44}^2$, 计算 F 在 $\mathbf{x}^{(2)}$ 处的

梯度，以下依次计算 F 在 $\mathbf{x}^{(2)}$ 处关于 x_{mn} 的偏导数，其中 $1 \leq m \leq 4, 1 \leq n \leq 4$ 。

$$\begin{aligned} \frac{\partial F}{\partial x_{22} | x_{22}=6} &= 2 \times x_{22} = 2 \times 6 = 12, & \frac{\partial F}{\partial x_{12} | x_{12}=2} &= \frac{\partial F}{\partial x_{21} | x_{21}=4} = \frac{\partial F}{\partial x_{11} | x_{11}=0} = 0 \\ \frac{\partial F}{\partial x_{13} | x_{13}=9} &= 2 \times x_{13} = 2 \times 9 = 18, & \frac{\partial F}{\partial x_{24} | x_{24}=0} &= \frac{\partial F}{\partial x_{14} | x_{14}=3} = \frac{\partial F}{\partial x_{23} | x_{23}=7} = 0 \\ \frac{\partial F}{\partial x_{42} | x_{42}=18} &= 2 \times x_{42} = 2 \times 18 = 36, & \frac{\partial F}{\partial x_{32} | x_{32}=13} &= \frac{\partial F}{\partial x_{41} | x_{41}=12} = \frac{\partial F}{\partial x_{31} | x_{31}=0} = 0 \\ \frac{\partial F}{\partial x_{44} | x_{44}=14} &= 2 \times x_{44} = 2 \times 14 = 28, & \frac{\partial F}{\partial x_{33} | x_{33}=1} &= \frac{\partial F}{\partial x_{34} | x_{34}=5} = \frac{\partial F}{\partial x_{43} | x_{43}=0} = 0 \end{aligned}$$

即 F 在 $\mathbf{x}^{(2)}$ 处的梯度，如图 12-6 所示。

$$\frac{\partial F}{\partial \mathbf{x} | \mathbf{x}=\mathbf{x}^{(2)}} =$$

0	0	18	0
0	12	0	0
0	0	0	0
0	36	0	28

图 12-6 F 在 $\mathbf{x}^{(2)}$ 处的梯度

根据标准的梯度下降法，结果如下：

$$\mathbf{x}^{(3)} = \mathbf{x}^{(2)} - \eta \frac{\partial F}{\partial \mathbf{x} | \mathbf{x}=\mathbf{x}^{(2)}}$$

展开后如图 12-7 所示。

$$\mathbf{x}^{(3)} =$$

0	2	9	3
4	6	7	0
0	13	1	5
12	18	0	14

$$- 0.5 \times$$

0	0	18	0
0	12	0	0
0	0	0	0
0	36	0	28

$$=$$

0	2	0	3
4	0	7	0
0	13	1	5
12	0	0	0

图 12-7 利用梯度下降法计算的第 2 次迭代值

依此类推。以上示例的对应代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
#"初始化x的值"
x=tf.Variable(tf.constant([
    [
        [8],[2],[9],[3]],
```

```

        [[4],[6],[7],[10]],
        [[20],[13],[1],[5]],
        [[12],[18],[19],[14]]
    ]
    ],tf.float32),dtype=tf.float32)

#"2×2的掩码，步长为2×2的最大值池化操作"
x_maxPool=tf.nn.max_pool(x,(1,2,2,1),(1,2,2,1),'VALID')
#"对以上最大值池化结果计算其平方和"
F=tf.reduce_sum(tf.square(x_maxPool))
#"创建会话"
session=tf.Session()
session.run(tf.global_variables_initializer())
#"梯度下降法"
opti=tf.train.GradientDescentOptimizer(0.5).minimize(F)
#"打印前2次的结果"
for i in range(2):
    session.run(opti)
    print(session.run(x))

```

打印结果如下：

```

[[
  [[ 0.],[ 2.],[ 9.],[ 3.]],
  [[ 4.],[ 6.],[ 7.],[ 0.]],
  [[ 0.],[13.],[ 1.],[ 5.]],
  [[12.],[18.],[ 0.],[14.]]
]]
[[
  [[ 0.],[ 2.],[ 0.],[ 3.]],
  [[ 4.],[ 0.],[ 7.],[ 0.]],
  [[ 0.],[13.],[ 1.],[ 5.]],
  [[12.],[ 0.],[ 0.],[ 0.]]
]]

```

打印结果与手动计算的 $\mathbf{x}^{(2)}$ 和 $\mathbf{x}^{(3)}$ 相等，修改以上梯度下降的迭代次数，由 2 次改为 4 次，就会得到最优值，对应的代码如下：

```

#"打印第4次的结果"

```



```

for i in range(4):
    session.run(opti)
    if(i==3):
        print(session.run(x))

```

打印结果如下：

```

[[
[[ 0.],[ 0.],[ 0.],[ 0.]],
[[ 0.],[ 0.],[ 0.],[ 0.]],
[[ 0.],[ 0.],[ 0.],[ 0.]],
[[ 0.],[ 0.],[ 0.],[ 0.]]
]]

```

显然，函数 F 在自变量 $x_{mn} = 0$ 处有最小值，其中 $1 \leq m \leq 4, 1 \leq n \leq 4$ 。以上示例中使用的是标准的梯度下降法，也可以使用第 3 章介绍的其他梯度下降法，代码实现中只要替换上述代码中的函数 `GradientDescentOptimizer` 即可。

10.5.2 节介绍了在卷积神经网络的卷积层使用 BN 操作，第 13 章将介绍如何在全连接层上使用 BN 操作，并介绍其对应的梯度反向传播。

BN 的梯度反向传播

13.1 BN 操作与卷积的关系

在第 11 章和第 12 章中，我们分别介绍了卷积神经网络中关于卷积和池化的梯度反向传播，本章我们介绍如果神经网络中有 BN 操作，该如何计算梯度。

在 10.5.2 节中，我们介绍了在卷积层的 BN 操作，本章我们介绍在全连接层的 BN 操作，如图 13-1 所示的神经网络，其中输入层、隐含层和输出层均包含两个神经元，隐含层有激活操作，其中激活函数记为 $f(\cdot)$ ，为了方便讨论，我们把隐含层的线性组合 $l^{(1)}$ 和对线性组合的激活 $\sigma^{(1)}$ 分开展示。

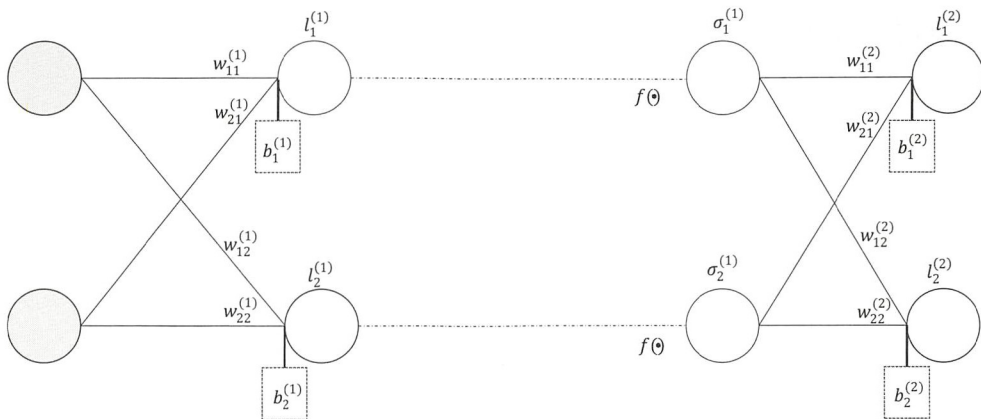


图 13-1 三层全连接神经网络

针对图 13-1 所示的神经网络，在隐含层的线性组合之后和激活操作之前加上 BN 操作，假设 $l_1^{(1)}$ 进行 BN 处理的结果记为 $z_1^{(1)}$ ， $l_2^{(1)}$ 进行 BN 处理的结果记为 $z_2^{(1)}$ ，如图 13-2 所示。

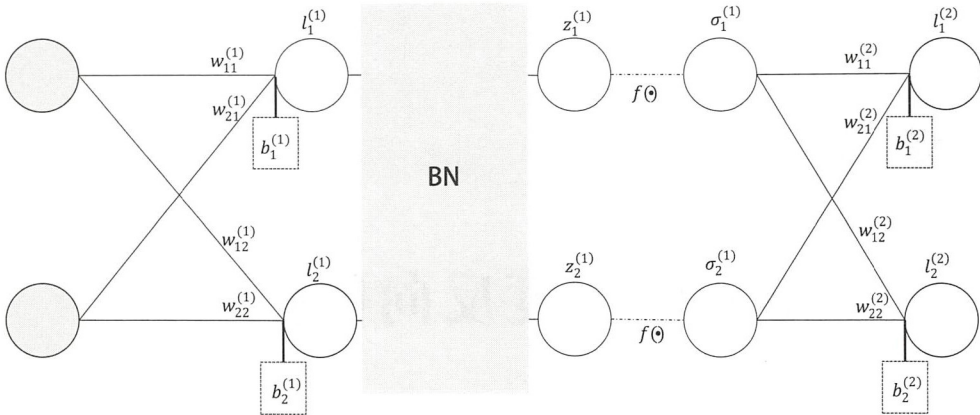


图 13-2 具有 BN 操作的全连接神经网络

在 10.5.3 节中，我们已经得知在卷积层的 BN 操作，相当于先与 1×1 的卷积核 `depthwise_conv2d` 卷积，接着与偏置相加，再与 1×1 的卷积核 `depthwise_conv2d` 卷积，最后与偏置相加，那么在全连接层中 BN 操作是不是也可以有另一种解释方式呢？我们对 x 的 BN 处理分为两步，第一步是标准化

$$\tilde{z} = \frac{x - \mu}{\sigma} = x \cdot \frac{1}{\sigma} + \left(-\frac{\mu}{\sigma}\right)$$

第二步是平移和缩放

$$z = \gamma \tilde{z} + \beta$$

上述两步可以用图 13-3 表示。

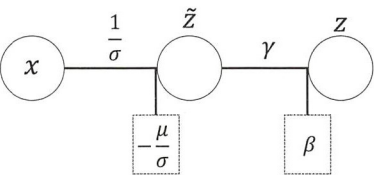


图 13-3 BN 操作

那么图 13-2 可以改进为图 13-4。

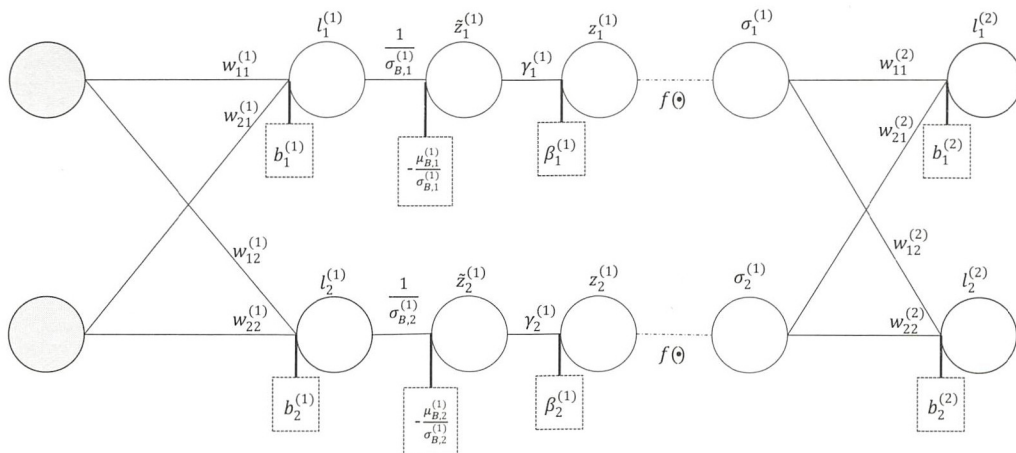


图 13-4 展开图 13-2 所示的 BN 操作的全连接神经网络

现在考虑一个问题, 假设有针对性图 13-4 所示的网络结构, 有一个已知的输入, 该输入经过网络后输出层的结果为 $l_1^{(2)}$ 、 $l_2^{(2)}$, 根据 $l_1^{(2)}$ 和 $l_2^{(2)}$ 构造一个可导的函数 F , 假设为

$$F = (l_1^{(2)} + 2l_2^{(2)})^2$$

同 6.4 节介绍的导数反向传播类似, F 针对图 13-4 所示的中间变量的偏导数可以用图 13-5 描述。

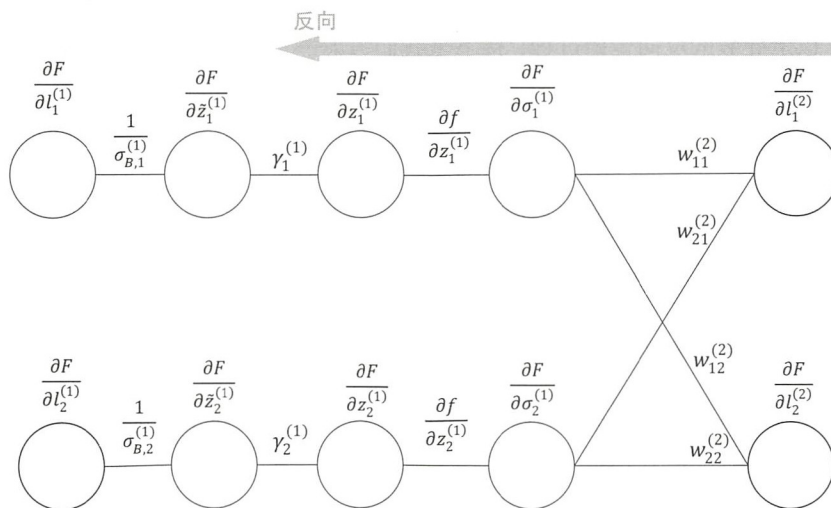


图 13-5 导数的反向传播

根据图 13-5 我们可以快速计算某一个中间变量的偏导数，比如我们计算 F 针对中间变量 $z_1^{(1)}$ 的偏导数 $\frac{\partial F}{\partial z_1^{(1)}}$ ，可以根据图 13-6 所示的后向传播依次进行计算。

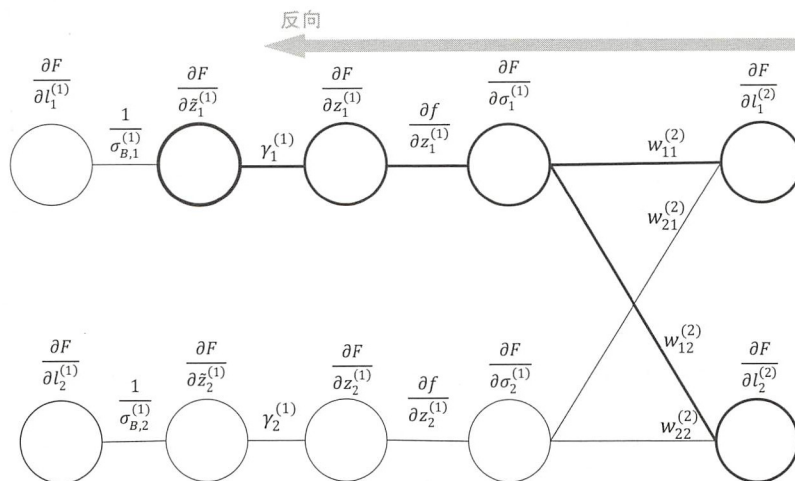


图 13-6 计算 F 针对中间变量 $z_1^{(1)}$ 的偏导数

13.2 示例详解

理解了以上导数的反向传播过程，接下来我们考虑一个更具体的问题，假设针对图 13-4 所示的网络结构有三个输入，分别为

$$\begin{bmatrix} 4 & 1 \end{bmatrix}, \begin{bmatrix} 3 & 3 \end{bmatrix}, \begin{bmatrix} 6 & 2 \end{bmatrix}$$

这三个输入经过网络后的输出分别记为

$$\begin{bmatrix} l_1^{(2)\{1\}} & l_2^{(2)\{1\}} \end{bmatrix}, \begin{bmatrix} l_1^{(2)\{2\}} & l_2^{(2)\{2\}} \end{bmatrix}, \begin{bmatrix} l_1^{(2)\{3\}} & l_2^{(2)\{3\}} \end{bmatrix}$$

根据以上三个输出层的值，分别构造三个函数：

$$\begin{aligned} & F^{(i)}(w_{11}^{(1)}, w_{21}^{(1)}, w_{12}^{(1)}, w_{22}^{(1)}, b_1^{(1)}, b_2^{(1)}, \gamma_1^{(1)}, \beta_1^{(1)}, \gamma_2^{(1)}, \beta_2^{(1)}, w_{11}^{(2)}, w_{21}^{(2)}, w_{12}^{(2)}, w_{22}^{(2)}, b_1^{(2)}, b_2^{(2)}) \\ &= (l_1^{(2)\{i\}} + 2l_2^{(2)\{i\}})^2 \end{aligned}$$

其中 $i = 1, 2, 3$ ，函数 F 是这 3 个函数的和，即

$$\begin{aligned}
& F(w_{11}^{(1)}, w_{21}^{(1)}, w_{12}^{(1)}, w_{22}^{(1)}, b_1^{(1)}, b_2^{(1)}, \gamma_1^{(1)}, \beta_1^{(1)}, \gamma_2^{(1)}, \beta_2^{(1)}, w_{11}^{(2)}, w_{21}^{(2)}, w_{12}^{(2)}, w_{22}^{(2)}, b_1^{(2)}, b_2^{(2)}) \\
&= \sum_{i=1}^3 F^{(i)}(w_{11}^{(1)}, w_{21}^{(1)}, w_{12}^{(1)}, w_{22}^{(1)}, b_1^{(1)}, b_2^{(1)}, \gamma_1^{(1)}, \beta_1^{(1)}, \gamma_2^{(1)}, \beta_2^{(1)}, w_{11}^{(2)}, w_{21}^{(2)}, w_{12}^{(2)}, w_{22}^{(2)}, b_1^{(2)}, b_2^{(2)}) \\
&= \sum_{i=1}^3 (l_1^{(2)\{i\}} + 2l_2^{(2)\{i\}})^2
\end{aligned}$$

我们计算 F 在点

$$\begin{aligned}
& (w_{11}^{(1)}, w_{21}^{(1)}, w_{12}^{(1)}, w_{22}^{(1)}, b_1^{(1)}, b_2^{(1)}, \gamma_1^{(1)}, \beta_1^{(1)}, \gamma_2^{(1)}, \beta_2^{(1)}, w_{11}^{(2)}, w_{21}^{(2)}, w_{12}^{(2)}, w_{22}^{(2)}, b_1^{(2)}, b_2^{(2)}) \\
&= (2, 3, 1, 4, -2, -3, 3, 6, 2, 4, 5, 2, 3, 8, -4, -6)
\end{aligned}$$

处的梯度。为了便于介绍，该点记为 \mathbf{p} ，因为函数和的梯度等于函数梯度的和，所以我们只要计算函数 $F^{(i)}$ 在 \mathbf{p} 处的梯度，然后求和，即为 F 在 \mathbf{p} 处的梯度。

接下来，我们利用导数的反向传播快速计算以上问题，先将第 1 个输入 $\begin{bmatrix} 4 & 1 \end{bmatrix}$ 输入图 13-4 所示的网络结构，然后计算其线性组合 $\mathbf{l}^{(1)\{1\}}$ ，其中 $\mathbf{l}^{(1)\{1\}} = \begin{bmatrix} l_1^{(1)\{1\}} & l_2^{(1)\{1\}} \end{bmatrix}$ ，其结果如图 13-7 所示。

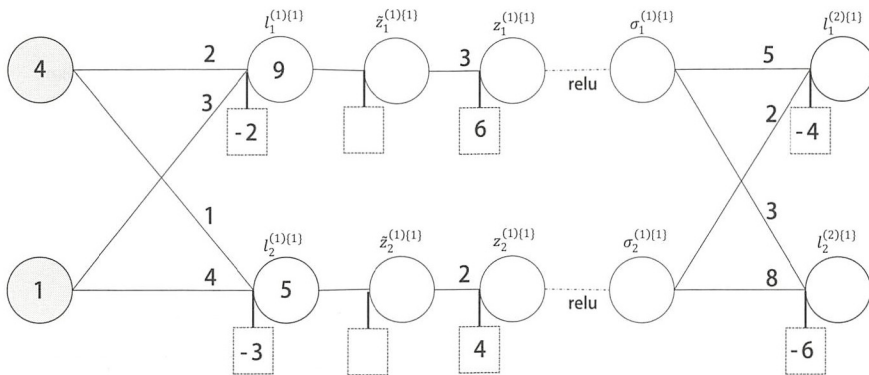


图 13-7 输入层的值是 $[4, 1]$

将第 2 个输入 $\begin{bmatrix} 3 & 3 \end{bmatrix}$ 输入图 13-4 所示的网络结构，计算其线性组合 $\mathbf{l}^{(1)\{2\}}$ ，其中 $\mathbf{l}^{(1)\{2\}} = \begin{bmatrix} l_1^{(1)\{2\}} & l_2^{(1)\{2\}} \end{bmatrix}$ ，其结果如图 13-8 所示。

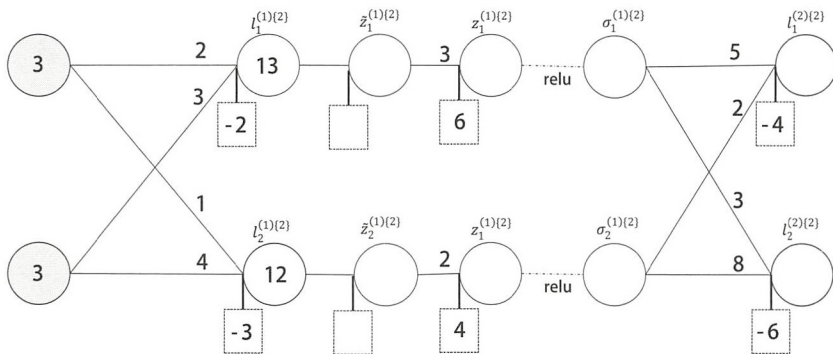


图 13-8 输入层的值是 [3, 3]

将第 3 个输入 $\begin{bmatrix} 6 \\ 2 \end{bmatrix}$ 输入图 13-4 所示的网络结构, 计算其线性组合 $l^{(1)\{3\}}$, 其中 $l^{(1)\{3\}} = \begin{bmatrix} l_1^{(1)\{3\}} \\ l_2^{(1)\{3\}} \end{bmatrix}$, 其结果如图 13-9 所示。

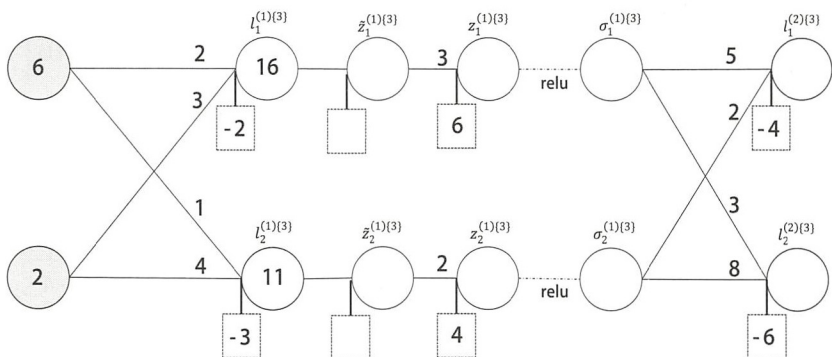


图 13-9 输入层的值是 [6, 2]

我们计算 $l_1^{(1)\{1\}}$ 、 $l_1^{(1)\{2\}}$ 、 $l_1^{(3)\{1\}}$ 这三者的平均值和标准差, 结果如下:

$$\mu_{B,1}^{(1)} = \frac{9 + 13 + 16}{3} = 12.7, \quad \sigma_{B,1}^{(1)} = \sqrt{\frac{(9 - 12.7)^2 + (13 - 12.7)^2 + (16 - 12.7)^2}{3}} = 2.87$$

同理, 计算 $l_2^{(1)\{1\}}$ 、 $l_2^{(1)\{2\}}$ 、 $l_2^{(3)\{1\}}$ 这三者的平均值和标准差, 结果如下:

$$\mu_{B,2}^{(1)} = \frac{5 + 12 + 11}{3} = 9.3, \quad \sigma_{B,2}^{(1)} = \sqrt{\frac{(5 - 9.3)^2 + (12 - 9.3)^2 + (11 - 9.3)^2}{3}} = 3.09$$

根据上述结果, 我们计算出了图 13-4 所示的网络结构中权重 $\frac{1}{\sigma_{B,1}^{(1)}}$ 、 $\frac{1}{\sigma_{B,2}^{(1)}}$, 偏置 $-\frac{\mu_{B,1}^{(1)}}{\sigma_{B,1}^{(1)}}$ 、 $-\frac{\mu_{B,2}^{(1)}}{\sigma_{B,2}^{(1)}}$, 则如图 13-7 ~ 图 13-9 所示, 可以接着进行前向计算, 其结果如图 13-10 ~ 图 13-12

所示。

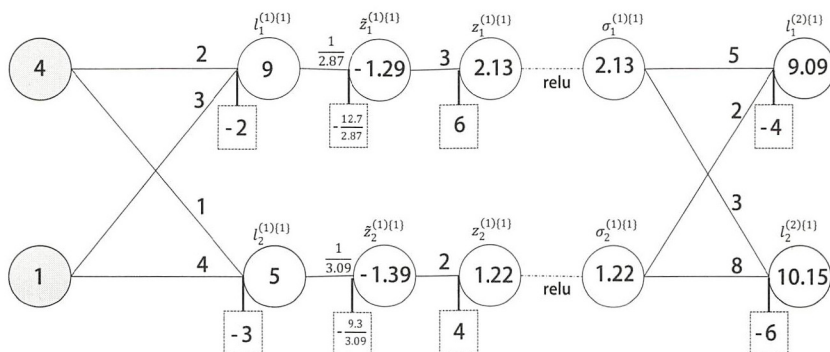


图 13-10 当输入层的值是 [4, 1] 时，计算所有神经元处的值

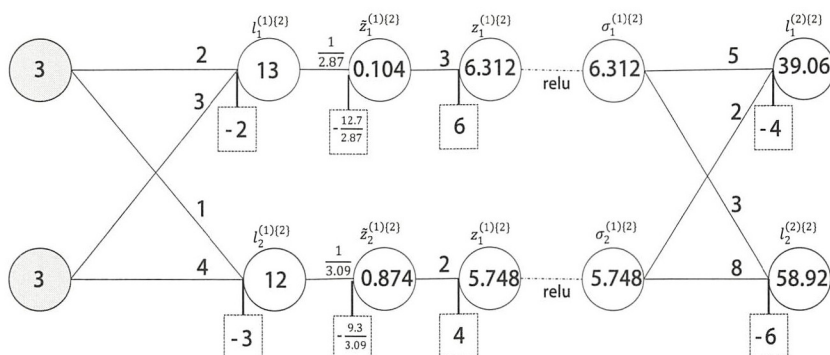


图 13-11 当输入层的值是 [3, 3] 时，计算所有神经元处的值

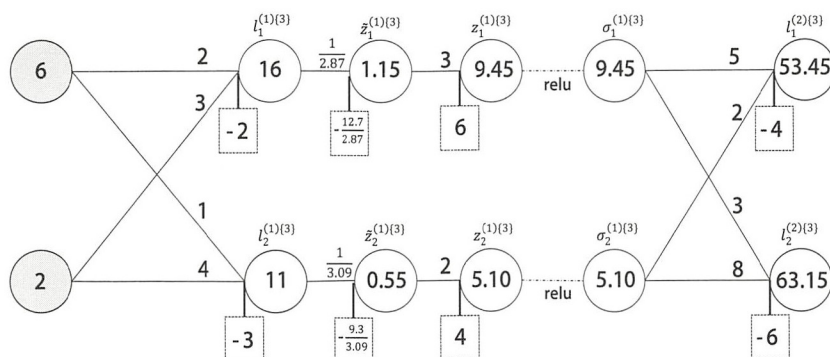


图 13-12 当输入层的值是 [6, 2] 时，计算所有神经元处的值

显然, $F^{(1)}$ 针对 $l_1^{(2)\{1\}}$, 在 $l_1^{(2)\{1\}} = 9.09$ 处的偏导数为

$$\frac{\partial F^{(1)}}{\partial l_1^{(2)\{1\}}} \Big|_{l_1^{(2)\{1\}}=9.09} = 2 \times (9.09 + 2 \times 10.15) = 58.78$$

$F^{(1)}$ 针对 $l_2^{(2)\{1\}}$, 在 $l_2^{(2)\{1\}} = 10.15$ 处的偏导数为

$$\frac{\partial F^{(1)}}{\partial l_2^{(2)\{1\}}} \Big|_{l_2^{(2)\{1\}}=10.15} = 2 \times (9.09 + 2 \times 10.15) \times 2 = 117.56$$

我们可以将上述结果看成类似于图 13-5 所示的导数反向传播的网络结构的输入层的值, 如图 13-13 所示。

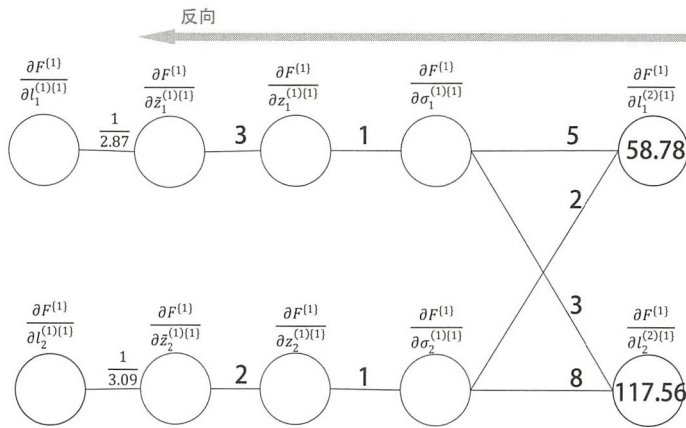


图 13-13 $F^{(1)}$ 在 $l_1^{(2)\{1\}} = 9.09$, $l_2^{(2)\{1\}} = 10.15$ 处的偏导数

接着, 按照从右向左的顺序依次计算 $F^{(1)}$ 关于其他中间变量的梯度, 如图 13-14 所示。

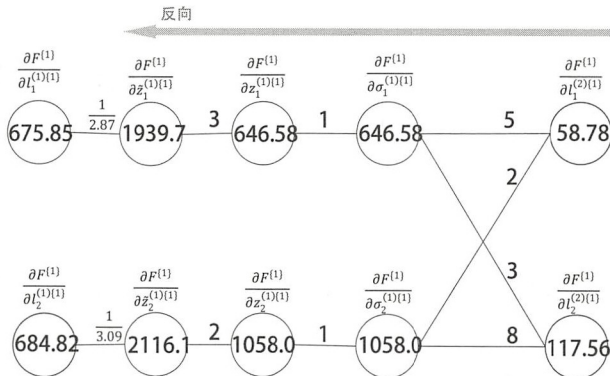


图 13-14 $F^{(1)}$ 关于其他中间变量的梯度

如图 13-10 和图 13-14 所示, 我们很容易得到 $F^{(1)}$ 针对中间变量 $\sigma_1^{(1)\{1\}}$ 和 $\sigma_2^{(1)\{1\}}$, 在 $\begin{bmatrix} \sigma_1^{(1)\{1\}} \\ \sigma_2^{(1)\{1\}} \end{bmatrix} = \begin{bmatrix} 2.13 \\ 1.22 \end{bmatrix}$ 处的梯度为

$$\frac{\partial F^{(1)}}{\partial \sigma^{(1)\{1\}}} \begin{bmatrix} \sigma_1^{(1)\{1\}} \\ \sigma_2^{(1)\{1\}} \end{bmatrix} = \begin{bmatrix} 646.58 \\ 1058.0 \end{bmatrix}$$

同理, $F^{(1)}$ 针对中间变量 $z_1^{(1)\{1\}}$ 和 $z_2^{(1)\{1\}}$, 在 $\begin{bmatrix} z_1^{(1)\{1\}} \\ z_2^{(1)\{1\}} \end{bmatrix} = \begin{bmatrix} 2.13 \\ 1.22 \end{bmatrix}$ 处的梯度为

$$\frac{\partial F^{(1)}}{\partial \mathbf{z}^{(1)\{1\}}} \begin{bmatrix} z_1^{(1)\{1\}} \\ z_2^{(1)\{1\}} \end{bmatrix} = \begin{bmatrix} 646.58 \\ 1058.0 \end{bmatrix}$$

$F^{(1)}$ 针对中间变量 $\tilde{z}_1^{(1)\{1\}}$ 和 $\tilde{z}_2^{(1)\{1\}}$, 在 $\begin{bmatrix} \tilde{z}_1^{(1)\{1\}} \\ \tilde{z}_2^{(1)\{1\}} \end{bmatrix} = \begin{bmatrix} -1.29 \\ -1.39 \end{bmatrix}$ 处的梯度为

$$\frac{\partial F^{(1)}}{\partial \tilde{\mathbf{z}}^{(1)\{1\}}} \begin{bmatrix} \tilde{z}_1^{(1)\{1\}} \\ \tilde{z}_2^{(1)\{1\}} \end{bmatrix} = \begin{bmatrix} 1939.7 \\ 2116.1 \end{bmatrix}$$

$F^{(1)}$ 针对中间变量 $l_1^{(1)\{1\}}$ 和 $l_2^{(1)\{1\}}$, 在 $\begin{bmatrix} l_1^{(1)\{1\}} \\ l_2^{(1)\{1\}} \end{bmatrix} = \begin{bmatrix} 9 \\ 5 \end{bmatrix}$ 处的梯度为

$$\begin{bmatrix} l_1^{(1)\{1\}} \\ l_2^{(1)\{1\}} \end{bmatrix} = \begin{bmatrix} 675.85 \\ 684.82 \end{bmatrix}$$

同理, $F^{(2)}$ 针对 $l_1^{(2)\{2\}}$, 在 $l_1^{(2)\{2\}} = 39.06$ 处的偏导数为

$$\frac{\partial F^{(2)}}{\partial l_1^{(2)\{2\}} \big|_{\mathbf{p}, l_1^{(2)\{2\}}=9.09}} = 2 \times (39.06 + 2 \times 58.92) = 313.78$$

$F^{(2)}$ 针对 $l_2^{(2)\{2\}}$, 在 $l_2^{(2)\{2\}} = 58.92$ 处的偏导数为

$$\frac{\partial F^{(2)}}{\partial l_2^{(2)\{2\}} \big|_{\mathbf{p}, l_2^{(2)\{2\}}=10.15}} = 2 \times (39.06 + 2 \times 58.92) \times 2 = 627.57$$

我们也可以将上述结果看成类似于图 13-3 所示的全连接神经网络输入层的值, 如图 13-15 所示。

接着, 计算 $F^{(2)}$ 关于其他中间变量的梯度, 如图 13-16 所示。

同理， $F^{(3)}$ 针对 $l_1^{(2)\{3\}}$ ，在 $l_1^{(2)\{3\}} = 53.45$ 处的偏导数为

$$\frac{\partial F^{(3)}}{\partial l_1^{(2)\{3\}}}\bigg|_{p, l_1^{(2)\{3\}}=53.45} = 2 \times (53.45 + 2 \times 63.15) = 359.54$$

$F^{(3)}$ 针对 $l_2^{(2)\{3\}}$ ，在 $l_1^{(2)\{3\}} = 63.15$ 处的偏导数为

$$\frac{\partial F^{(3)}}{\partial l_2^{(2)\{3\}}}\bigg|_{p, l_2^{(2)\{3\}}=63.15} = 2 \times (53.45 + 2 \times 63.15) \times 2 = 719.09$$

我们同样可以将上述结果看成类似于图 13-5 所示的导数反向传播的网络结构的输入层的值，如图 13-17 所示。

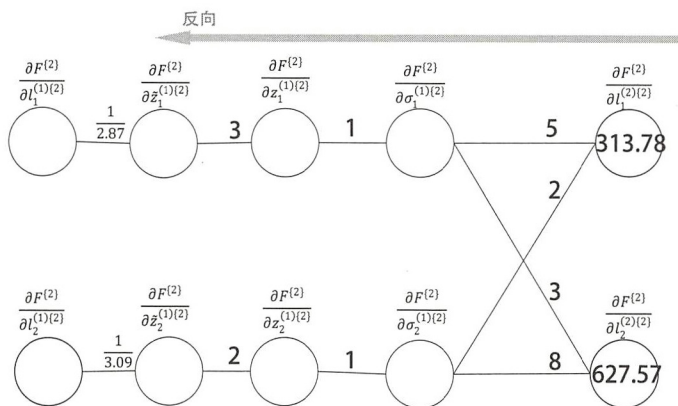


图 13-15 $F^{(2)}$ 在 $l_1^{(2)\{2\}} = 39.06$ ， $l_2^{(2)\{2\}} = 58.92$ 处的偏导数

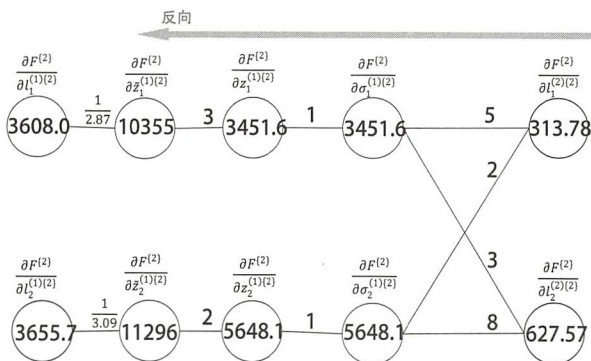
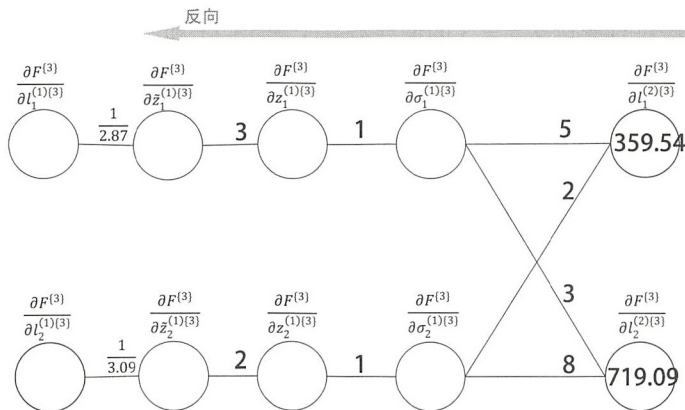
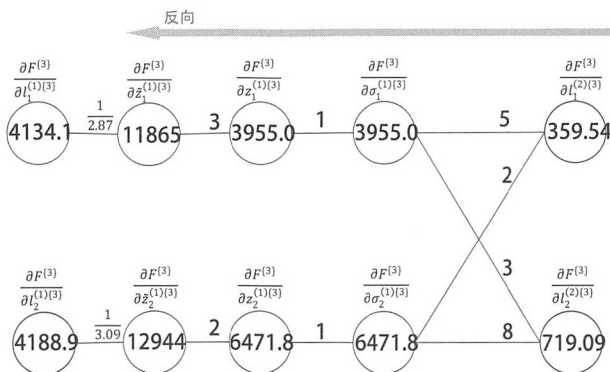


图 13-16 $F^{(2)}$ 关于其他中间变量的梯度

图 13-17 $F^{(3)}$ 在 $l_1^{(2)(3)} = 53.45$, $l_2^{(2)(3)} = 63.15$ 处的偏导数

接着，计算 $F^{(3)}$ 关于其他中间变量的梯度，如图 13-18 所示。

图 13-18 $F^{(3)}$ 关于其他中间变量的梯度

我们的目标是计算 F 在点 \mathbf{p} 处的梯度，那么只要计算 $F^{(i)}$ 在 \mathbf{p} 处的梯度即可，我们以计算在 \mathbf{p} 处针对自变量 $\gamma_1^{(1)} = 3$ 的偏导数为例，因为

$$\frac{\partial F^{(i)}}{\partial \gamma_1^{(1)}} = \frac{\partial F^{(i)}}{\partial z_1^{(1)(i)}} \frac{\partial z_1^{(1)(i)}}{\partial \gamma_1^{(1)}} = \frac{\partial F^{(i)}}{\partial z_1^{(1)}} \tilde{z}_1^{(i)}, \quad i = 1, 2, 3$$

所以， $F^{(1)}$ 在 \mathbf{p} 处针对自变量 $\gamma_1^{(1)} = 3$ 的偏导数等于图 13-10 所示的 $\tilde{z}_1^{(1)(1)}$ 处的值和图 13-14 所示的 $\frac{\partial F^{(1)}}{\partial z_1^{(1)(1)}}$ 处的值相乘，即

$$\frac{\partial F^{(1)}}{\partial \gamma_1^{(1)}} \Big|_{\gamma_1^{(1)}=3} = 646.58 \times (-1.29) = -834.0882$$

$F^{(2)}$ 在 \mathbf{p} 处针对自变量 $\gamma_1^{(1)} = 3$ 的偏导数等于图 13-11 和图 13-16 所示的 $\frac{\partial F^{(2)}}{\partial z_1^{(1)(2)}}$ 处的值相乘，即

$$\frac{\partial F^{(2)}}{\partial \gamma_1^{(1)} |_{\gamma_1^{(1)}=3}} = 3451.6 \times 0.104 = 358.966$$

$F^{(3)}$ 在 \mathbf{p} 处针对自变量 $\gamma_1^{(1)} = 3$ 的偏导数等于图 13-12 和图 13-18 所示的 $\frac{\partial F^{(3)}}{\partial z_1^{(1)(3)}}$ 处的值相乘，即

$$\frac{\partial F^{(3)}}{\partial \gamma_1^{(1)} |_{\gamma_1^{(1)}=3}} = 3955 \times 1.15 = 4548.25$$

那么

$$\frac{\partial F}{\partial \gamma_1^{(1)} |_{\gamma_1^{(1)}=3}} = -834.0882 + 358.966 + 4548.25 = 4073.1278$$

上述示例对应的代码如下：

```
# -*- coding: utf-8 -*-
import tensorflow as tf
import numpy as np
#""
x=tf.placeholder(tf.float32,[None,2])
#"权重和偏置"
w1=tf.Variable(tf.constant([
                                [2,1],
                                [3,4]
                                ],tf.float32))
b1=tf.Variable(tf.constant([-2,-3],tf.float32))
#"线性组合"
l1=tf.matmul(x,w1)+b1
#"对线性组合进行BN处理"
mu_B,var_B=tf.nn.moments(l1,0)
gamma1=tf.Variable(tf.constant([3,2],tf.float32))
beta1=tf.Variable(tf.constant([6,4],tf.float32))
z1=tf.nn.batch_normalization(l1,mu_B,var_B,beta1,gamma1,1e-8)
#"激活"
sigma1=tf.nn.relu(z1)
#"权重和偏置"
```

```

w2=tf.Variable(tf.constant([
                                [5,3],
                                [2,8]
                            ],tf.float32))
b2=tf.Variable(tf.constant([-4,-6],tf.float32))
#"线性组合"
l2=tf.matmul(sigma1,w2)+b2
#"构造函数F"
F=tf.reduce_sum(
    tf.pow(tf.reduce_sum(tf.multiply(l2,tf.constant([1,2],tf.float32)),1),2.0)
)
#"对gamma1的偏导数"
gamm1_gra=tf.gradients(F,[gamma1])
session=tf.Session()
session.run(tf.global_variables_initializer())
gamm1_gra=session.run(gamm1_gra,feed_dict={x:np.array(
    [
        [4,1],
        [3,3],
        [6,2]
    ],np.float32
    )})

#"打印结果"
print(gamm1_gra[0][0])

```

打印结果如下：

```
4173.5205
```

注意：程序的打印结果和手动计算的结果不相等。这是因为在手动计算的过程中多次四舍五入计算导致误差积累，使结果不是很准确。函数 F 在点 p 处的其他自变量的偏导数计算方式类似。

以上是在全连接层中使用 BN 处理，把 BN 处理看成如图 13-13 所示的线性组合，更容易理解梯度反向传播。因为在卷积神经网络中的卷积层中使用 BN 处理，相当于卷积运算，可以利用第 10 章的理论进行解释，这里不再赘述。

利用梯度下降法训练卷积神经网络时，快速计算损失函数在某点的梯度的核心思想是利用第 11 章和第 12 章介绍的关于卷积和池化操作的梯度算法。至此，我们比较全面地介绍了全连接神经网络及卷积神经网络。第 14 章将针对 TensorFlow 搭建这两类网络做一个总结。

14

TensorFlow 搭建神经网络的主要函数

本章主要梳理利用 TensorFlow 搭建神经网络的整个过程中使用的主要函数。

第 1 步：网络的输入。神经网络的输入可以利用占位符声明。

占位符

```
tf.placeholder(dtype, shape=None, name=None)
```

第 2 步：搭建网络。如果搭建的是全连接神经网络，则主要使用矩阵乘法和加法的函数。

矩阵乘法函数

```
tf.matmul(a, b, transpose_a=False, transpose_b=False, adjoint_a=False,  
          adjoint_b=False, a_is_sparse=False, b_is_sparse=False, name=None)
```

加法函数

```
tf.add(x, y, name=None)
```

如果搭建的是卷积神经网络，则主要使用卷积函数、池化函数和加法函数。

卷积函数

```
tf.nn.conv2d(input, filter, strides, padding, use_cudnn_on_gpu=True,  
             data_format='NHWC', dilations=[1, 1, 1, 1], name=None)
```

池化函数

```
tf.nn.max_pool(value, ksize, strides, padding, data_format='NHWC', name=None)
tf.nn.avg_pool(value, ksize, strides, padding, data_format='NHWC', name=None)
```

加法函数和搭建全连接神经网络使用的加法函数相同。当然，可以直接使用加法运算符“+”。

为了防止过拟合，可以添加 dropout 操作，对应的函数如下：

```
tf.nn.dropout(x, keep_prob, noise_shape=None, seed=None, name=None)
```

为了加快收敛速度，可以添加 BN 操作。注意，BN 操作常用在激活操作之前，对应的函数如下：

```
tf.moments(x, axes, shift=None)
tf.nn.batch_normalization(x, mean, variance, offset, scale, variance_epsilon)
```

搭建神经网络使用了如下激活函数。

```
tf.nn.sigmoid(x, name=None)
tf.nn.tanh(x, name=None)
tf.nn.relu(features, name=None)
tf.nn.leaky_relu(features, alpha=0.2, name=None)
tf.nn.relu6(features, name=None)
tf.nn.crelu(features, name=None)
tf.nn.selu(features, name=None)
tf.nn.softplus(features, name=None)
r=tf.nn.softsign(features, name=None)
```

第 3 步：创建损失函数。根据第 2 步搭建的网络，利用网络的输出值构造损失函数，常用的 3 个函数如下。

```
tf.reduce_sum(input_tensor, axis=None, keepdims=None, name=None,
               reduction_indices=None, keep_dims=None)
tf.nn.sigmoid_cross_entropy_with_logits(abels=None, logits=None)
tf.nn.softmax_cross_entropy_with_logits_v2(abels=None, logits=None)
```

第 4 步：选择优化器（即梯度下降法）。根据第 3 步构造的损失函数，针对它利用梯度下降法，常用的梯度下降函数有：

```
tf.train.GradientDescentOptimizer
tf.train.AdagradOptimizer
tf.train.MomentumOptimizer
tf.train.RMSPropOptimizer
tf.train.AdadeltaOptimizer
tf.train.AdamOptimizer
```

第 5 步：评估模型的准确率。评估模型的准确率就是对比人工分类和模型分类，所以使用的是对比函数：

```
tf.equal(x, y, name=None)
```

使用对比函数，返回值是 bool 型，所以需要类型转换函数：

```
tf.cast(x, dtype, name=None)
```

然后根据 cast 的返回值，利用平均值函数 `tf.reduce_mean(input_tensor, axis=None)` 计算准确率（百分比）。

第 6 步：保存和加载模型。训练模型结束后，需要保存模型，创建类 `tf.train.Saver`，然后利用其成员函数 `save()` 保存模型。

模型加载时，利用其成员函数 `restore()` 加载到内存中。

以上 6 个步骤是利用 TensorFlow 搭建神经网络的主要过程，对应的函数也可以在前面的章节中找到详细的示例。

作者简介



张平

数学与应用数学专业，算法工程师。主要从事图像算法研究和产品的应用开发，以及有关机器学习、人工智能的应用研发工作。

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

下载资源：本书提供示例代码资源文件，可在【下载资源】处下载。

提交勘误：您对书中内容的修改意见可在【提交勘误】处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。

与读者交流：在页面下方【读者评论】处留下您的疑问或观点，与其他读者一同学习交流。

页面入口：<http://www.broadview.com.cn/34745>



博文视点Broadview



@博文视点Broadview



责任编辑：郑柳洁
封面设计：吴海燕

反馈意见或投稿

邮箱：zhenglj@phei.com.cn

微信号：Alinamercy

上架建议：计算机>人工智能

ISBN 978-7-121-34745-0



9 787121 347450 >

定价：79.00元